VISUALIZATION OF SENSOR NETWORK APPLICATIONS IN SIMULATED ENVIRONMENTS

by

SAMUEL BENNY KUMMARY

B.E., Andhra University, India, 2005

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2009

Approved by:

Major Professor
Dr. Gurdip Singh

Abstract

Distributed applications that operate on networks of sensors to gather data are important in real world. TinyOS is an operating system designed to support wireless sensor networks. . It has interfaces and components, which provide functionalities for sensing parameters in the environment, packet communication and computation. These sensors have multiple purposes such as gathering different kinds of data and can be deployed in distributed networks to gather important information.

NesC is a language which is used to write sensor applications for TinyOS which are deployment on the sensors. TinyViz is an application which simulates the NesC applications on a computer so that the applications can be tested first in the simulation environment and then can be tested on the sensors and deployed.

However, TinyViz by default represents a static and closed environment where the conditions simulated may not be realistic. This project aims at providing real-world scenarios on the platform TinyViz, by communicating with TinyViz using Tython, a script language for this specific purpose. In terms of sensor network applications, events are classified into categories, which can be mapped to tangible parameters. This project takes as input the real-world parameters as input by the developer of the NesC applications in the form of a configuration file and converts them into implementable threads that run in parallel with TinyViz and keep sending instructions to the TinyViz which then simulates real-world environment. Thus, it helps simulate NesC applications in a realistic environment even before the real deployment. This is packaged as an Eclipse plug-in for portability and ease of implementation, using which developers of NesC applications can give as input configuration and obtain the files required for simulation. The implementation is done in java, using ‘Tython’.

Table of Contents

List of Figures

Acknowledgements

Firstly, I would like to thank the God almighty, who is responsible for everything good.

Most importantly, I would like to express my gratitude to my advisor Dr. Gurdip Singh, who has supported me throughout my thesis and the entire MS program. I am thankful to Dr. Singh for supporting me in my low times, and for his patience and guidance, which made this possible.

I thank my committee members, Dr. Torben Amtoft and Dr. Mitchell L. Neilsen for their support, suggestions and valuable time. I would like to thank Dr. Daniel Andresen for giving me funding and opportunity to work during my first year.

Finally, I thank my mother and sister Mercy K, my friend Kala L, my family, my friends Sandeep Pulluri, Dinesh Challa, Pavani A, Phaninder S, Vikram K, Rohit P, Arka C, Deep G, Lakshman K, Ashok V, Sumeet G for their unrelenting support, constant encouragement and patience.

Dedication

I would like to dedicate this work to my advisor Dr. Gurdip Singh, who has been extremely supportive and for his guidance without which this wouldn’t be possible.

CHAPTER 1 – Introduction

1.1 Introduction

There are numerous real-world applications in which sensors are used. Generally they are used in applications to gather and report critical information. With advances in technology, there is an increased use in sensors in all walks of life to extract more information for the advancement of man-kind not just in terms of safety, comfort but also commercially. As the spread and use of sensors grows, the applications that are written for them become more diverse.

Hence, there is a need to make this process of developing these applications easier by using tools for automation and simulation. This project makes an effort to increase efficiency by simulating real-world scenarios which include calamities and other natural phenomenon. This simulation happens concurrently and hence, the behavior of the sensor applications in such conditions can be evaluated using this simulation.

1.2 TinyOS Applications and Real-world environment

TinyOS applications can be used in all places where the sensors are used. Most TinyOS applications deal with formation of networks, gathering data, communication of messages and hence, these applications cover an entire range of areas of application.

When working with embedded devices, it is very difficult to debug applications and also testing directly on the hardware. This necessitates use of simulation and other tools that minimize this debugging and testing effort. There are various tools that are discussed in the later part which help this cause and this project aims to make this even more relevant in testing the sensor applications.

The real-world has many situations which can be defined in physical parameters. In terms of sensor network applications, events in nature such as a forest fire occurring might be described as analogous to a wave of rise in temperature spreading in the direction of fire, followed by a wave of fall in temperature that represents the extinguishing of fire; similarly a tornado might be mapped to a dynamic random movement of a high pressure region; similarly diffusion of a gas mapped to raise or fall in one of the tangible parameters.

Thus, each of these situations can be analyzed and mapped to a physical state of tangible parameters; this enables us to define the physical situation in terms of numerical values and strings. This data can be used as background information and if a simulation can be provided which takes these values and applies them to the testing platform, it simulates the actual natural condition which this data represents. Thus, the simulation of real-world environment on the testing platform is made possible. The developers of sensor applications need to however, follow a protocol to define these conditions and give data for each specific situation to be simulated.

CHAPTER 2 – TinyOS, Tossim and Tython

2.1 TinyOS

2.2.1 Introduction to TinyOS

TinyOS is an open-source operating system designed for wireless embedded sensor networks. It features a component-based architecture which enables rapid innovation and implementation while minimizing code size as required by the severe memory constraints inherent in sensor networks. TinyOS's component library includes network protocols, distributed services, sensor drivers, and data acquisition tools – all of which can be used as-is or be further refined for a custom application. Many of these tools can be customized and modified to suit many real-world sensor applications.

TinyOS follows an event-driven execution model. This model enables fine-grained power management, yet allows the scheduling flexibility made necessary by the unpredictable nature of wireless communication and physical world interfaces. The TinyOS system, libraries, and applications are written in NesC, a language for programming structured component-based applications.

2.2.2 NesC

The NesC language is primarily intended for embedded systems such as sensor networks. NesC supports the TinyOS concurrency model, as well as mechanisms for structuring, naming, and linking together software components into robust network embedded systems. The principal goal is to allow application designers to build components that can be easily composed into complete, concurrent systems, and yet perform extensive checking at compile time.

NesC is an extension to the C programming language designed to embody the structuring concepts and execution model of TinyOS. TinyOS is an event-driven operating system designed for sensor network nodes that have very limited resources (e.g., 8K bytes of program memory, 512 bytes of RAM). The basic concepts behind

NesC are: separation of construction and composition, specification of component behavior in terms of set of interfaces.

A NesC application consists of one or more components linked together to form an executable. A component provides and uses interfaces. These interfaces are the only point of access to the component and are bi-directional. An interface declares a set of functions called commands that the interface provider must implement and another set of functions called events that the interface user must implement. For a component to call the commands in an interface, it must implement the events of that interface. A single component may use or provide multiple interfaces and multiple instances of the same interface. Interfaces may be provided or used by components. The provided interfaces are intended to represent the functionality that the component provides to its user; the 'used' interfaces represent the functionality the component needs to perform its job.

There are two types of components in NesC: modules and configurations. Modules provide application code, implementing one or more interface. Configurations are used to assemble other components together, connecting interfaces used by components to interfaces provided by others. This is called wiring. Every NesC application is described by a top-level configuration that wires together the components inside.

2.2.3 Execution environment in TinyOS

TinyOS executes only one program consisting of selected system components and custom components needed for a single application. There are two threads of execution: tasks and hardware event handlers. Tasks are functions whose execution is deferred. Once scheduled, they run to completion and do not preempt one another. Hardware event handlers are executed in response to a hardware interrupt and also run to completion, but may preempt the execution of a task or other hardware event handler. Commands and events that are executed as part of a hardware event handler must be declared with the 'async' keyword.

Components are statically linked to each other via their interfaces. This increases runtime efficiency, encourages robust design, and allows for better static analysis of programs. NesC is designed under the expectation that code will be generated by whole-program compilers. This should also allow for better code generation and analysis.

2.2.4 Programming with TinyOS

TinyOS applications are written in NesC, a dialect of the C programming language optimized for the memory limitations of sensor networks. Its supplementary tools are mainly in the form of Java and shell script front-ends. TinyOS programs are built out of software components, some of which present hardware abstractions. Components are connected to each other using interfaces.

TinyOS provides interfaces and components for common abstractions such as packet communication, routing, sensing, actuation and storage. TinyOS is completely non-blocking: it has a single stack. Non-blocking enables TinyOS to maintain high concurrency with a single stack, it requires writing many small event handlers to handle these events. TinyOS provides tasks, which are similar to a Deferred Procedure Call. A TinyOS component can post a task, which the OS will schedule to run later. Tasks are non-preemptive and run in FIFO order. TinyOS code is statically linked with program code, and compiled into a small binary, using a custom GNU tool-chain. Associated utilities are provided to complete a development platform for working with TinyOS.

2.2 TOSSIM

TOSSIM, the TinyOS simulator, compiles directly from TinyOS code and is a discrete event simulator for TinyOS sensor networks. Built with “make pc”, the simulation runs natively on a desktop or laptop. TOSSIM can simulate thousands of nodes simultaneously. Every mote in a simulation runs the same TinyOS program.

2.3.1 TOSSIM features and advantages

TOSSIM’s primary goal is to provide a high fidelity simulation of TinyOS applications. For this reason, it focuses on simulating TinyOS and its execution, rather than simulating the real world. Instead of compiling a TinyOS application for a mote, users can compile it into the TOSSIM framework, which runs on a PC. This allows users to debug, test, and analyze algorithms in a controlled and repeatable environment.

As TOSSIM runs on a PC, users can examine their TinyOS code using debuggers and other development tools. TOSSIM provides run-time configurable debugging output, allowing a user to examine the execution of an application from different perspectives without needing to recompile.

2.3.2 TOSSIM Limitations

While TOSSIM can be used to understand the causes of behavior observed in the real world, it does not capture all of them, and should not be used for absolute evaluations. TOSSIM is not always the right simulation solution; like any simulation, it makes several assumptions, focusing on making some behaviors accurate while simplifying others.

Experimenting with, and testing sensor networks is hard. TOSSIM, a TinyOS simulator, allows users to run and test algorithms, protocols, and applications in a controlled, reproducible environment. However, by itself a TOSSIM simulation is static. Instead of modeling behaviors such as motion or changing sensor readings, TOSSIM provides a socket-based command API for other programs to do so. On one hand, this keeps TOSSIM simple and efficient; on the other, it puts the burden of writing complex real-world models on the user.

2.3 TinyViz

3.2.1 TinyViz Introduction

TinyViz is a Java-based GUI that allows visualizing and controlling the simulation as it runs, inspecting debug messages, radio and UART packets, and communicates with TOSSIM over the socket API. The simulation provides several mechanisms for interacting with the network; packet traffic can be monitored, packets can be statically or dynamically injected into the network.

The main TinyViz class is a jar file, tools/java/net/tinyos/sim/tinyviz.jar. TinyViz can be attached to a running simulation. TinyViz is not actually a visualizer; instead, it is a framework in which plug-ins can provide desired functionality.

3.2.2 TinyViz Advantages

With TinyViz, users can interact with a simulation through a GUI panel, by dragging motes and setting options. These actions can be difficult to reproduce exactly (e.g., dragging a mote). Additionally, TinyViz can (as its name suggests) visualize what goes on in the network. Users can write TinyViz ``plug-ins" in Java to extend the GUI's functionality and issue commands to TOSSIM. However, because users must pre-compile their plug-ins, this only allows limited interactivity. Also, TOSSIM can be made to wait for TinyViz to connect before it starts up, with the -gui flag. This allows users to be sure that TinyViz captures all of the events in a given simulation.

TinyViz can use physical topologies to generate network topologies by sending messages to TOSSIM that configure network connectivity and the loss rate of individual links. TinyViz can slow a simulation by introducing delays when it handles events from TOSSIM. The slider configures how long delays are. The TinyViz engine uses an event-driven model, which allows easy mapping between TinyOS' event-based execution and event-driven GUIs. By itself, the application does very little; drop-in plug-ins provides user functionality. TinyViz has an event bus, which reads events from a simulation and publishes them to all active plug-ins.

By itself, TinyViz does little besides draw motes and their LEDs. However, it comes with a few example plug-ins, such as one that visualizes network traffic. Using TinyViz, you can easily trace the execution of TinyOS apps, set breakpoints when interesting events occur, visualize radio messages, and manipulate the virtual position and radio connectivity of motes.

2.4 Tython Introduction

Tython is a new system that adds a powerful new tool to the sensor network developer's portfolio. Through both predefined scripts and interactive console sessions, Tython aids the tasks of developing, testing, and evaluating a new algorithm or application. The core architecture is extensible, allowing developers to write new python modules and SimDriver plug-ins to add new forms of interaction and manipulation.

Tython (or, Tinython) complements the visualization of TinyViz by adding a scripting interface to TOSSIM. Users can interact with a running simulation through TinyViz, a Tython console, or both simultaneously. Tython is based on Jython, a Java implementation of the Python language. In addition to being a complete scripting language, Jython makes it very easy to import and use Java classes within Python. This allows users to access the entire TinyOS Java tool chain, including packet sources, MIG-generated messages, and TinyViz. TinyViz and Tython sit on top of SimDriver, a java application that manages interactions with TOSSIM. The figure below, give an insight into how the TinyViz and TOSSIM are related.

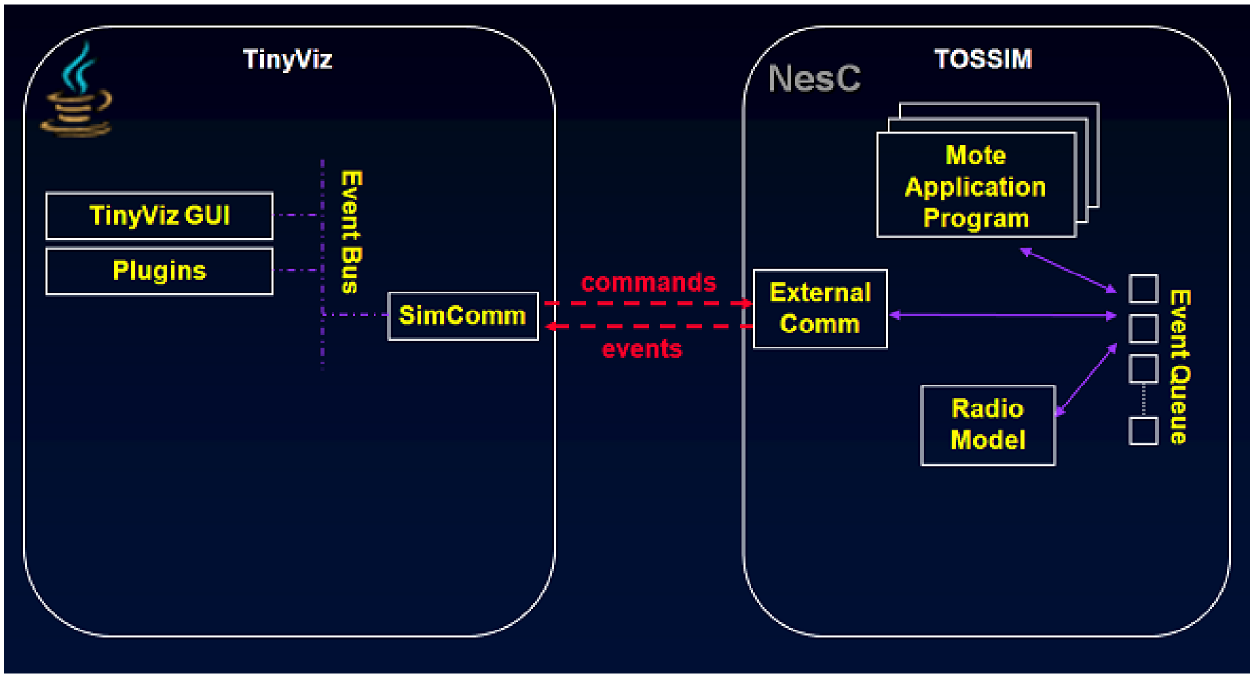

Fig. 1 – TinyViz and TOSSIM

The primary goal of Tython is to offer the sensor network developer a simulation environment with dynamic interactivity, enabling both unattended simulation experiments, as well as interactive debugging and simulation control. The confluence of these two goals informs the major design decisions of our project, as well as the particular interfaces exposed by the Tython commands and classes. TOSSIM essentially just simulates tossing a set of motes into a field and letting them go, assuming a constant radio topology. On the other hand, the real world is a dynamic place; objects and motes can move, radio connectivity changes, motes can fail. An important tool in the developer's toolbox, therefore, is the ability to simulate these dynamic interactions and thereby engineer a program that can cope with these situations. The figure below explains the architecture of communication between TOSSIM, TinyViz GUI and SimDriver and indicates where scripting fits in.

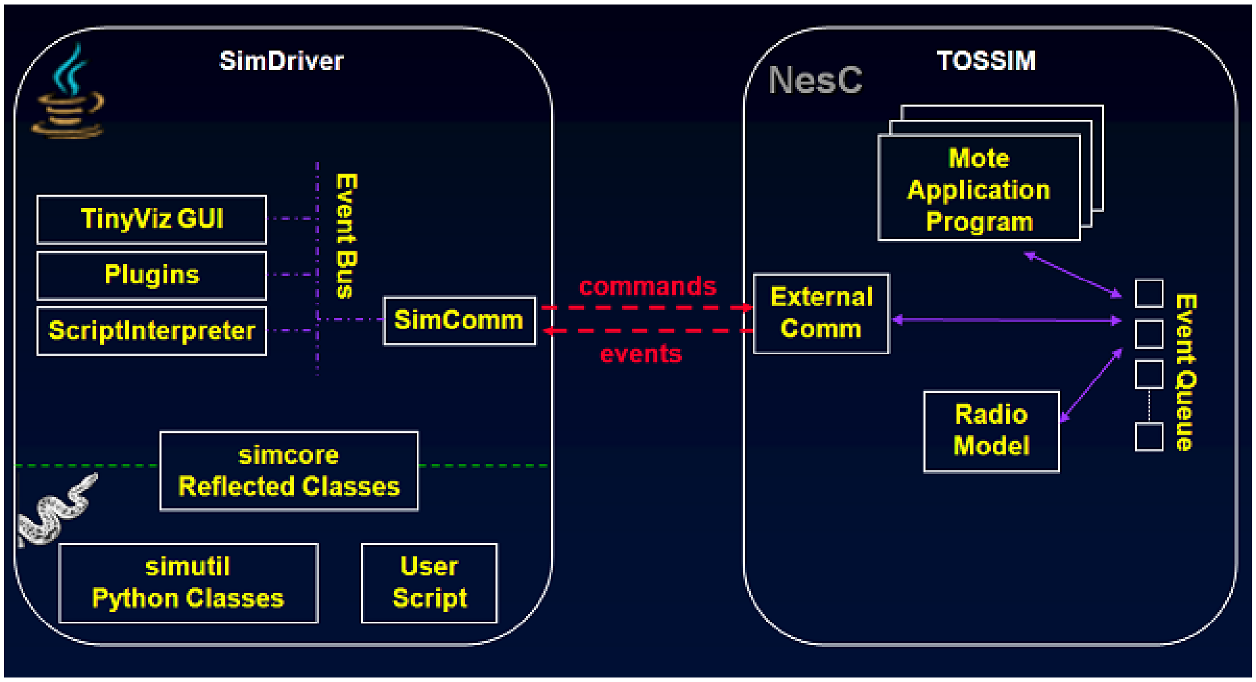

Fig. 2 – TOSSIM, Tython and scripting

TinyViz was a tool that enables developers to dynamically manipulate a simulation. The protocol between TOSSIM and TinyViz enables the GUI to introduce dynamics into a test application's execution. In point of fact, much of the core Tython functionality is implemented using the TinyViz plug-in system. Through a scripting environment, developers are able to control experiments through repeatable interactions. Because of the features provided by this framework, a developer can pause a simulation at a given time, use the variable resolution features to probe around (and potentially alter) the simulation state, then continue the simulation to observe the effects of the actions.

2.5 Using Tython to control Simulation

The simplest way to use Tython is to start a TOSSIM simulation with the -gui option, then run SimDriver with the -console option:

java net.tinyos.sim.SimDriver –console

Tython has several objects that provide functions to interact with TOSSIM which specifically can be used to execute, pause and resume a simulation. The commands are shown below:

- *sim.pause()* Pause TOSSIM
- *sim.resume()* Resume TOSSIM

2.6 Basic Simulation Commands

Various features in the form of commands are provided by Tython, which may either be used in scripts that may be run by running the script files or directly at the prompt. Some example commands and their effect are given below:

- motes[i].turnOn()

  Turn mote i on
- motes[i].turnOff()

  Turn mote i off
- motes[i].moveTo(x, y)

  Move mote i to x,y
- motes[i].move(x,y)

  Move mote i at x,y from its current position
- comm.setSimRate(rate)

  Set the simulator rate(This is identical to TOSSIM's -l option)
- radio.setLossRate(senderID, receiverID, prob)

  Set the radio loss rate between two motes
- comm.sendRadioMessage(mote, time, message)

  Deliver message to mote over Radio
- comm.sendUARTMessage(mote, time, message)

  Deliver message to mote over UART
- comm.setADCValue(moteID, time, port, value)

  Set the ADC value at the given mote to the specified value.

CHAPTER 3 – Simulation Overview and Architecture

3.1 Simulation Overview

Debugging and testing sensor networks applications by deploying on the hardware takes up good amount of time and effort; moreover testing and debugging sensor network applications is hard. TOSSIM, a TinyOS simulator, allows users to run and test algorithms, protocols, and applications in a controlled, reproducible environment.

However, by itself a TOSSIM simulation is static. Instead of modeling behaviors such as motion or changing sensor readings, TOSSIM provides a socket-based command API for other programs to do so. Hence, TOSSIM in conjunction with TinyViz which in-turn uses SimDriver is used. This with scripting provides the developer of sensor network applications effective tools to develop, test and simulate NesC applications before deployment and also automates testing.

One solution to this problem is TinyViz, a GUI that communicates with TOSSIM over the socket API. With TinyViz, users can interact with a simulation through a GUI panel, by dragging motes and setting options. These actions can be difficult to reproduce exactly (e.g., dragging a mote). Additionally, TinyViz can (as its name suggests) visualize what goes on in the network. Users can write TinyViz ``plug-ins" in Java to extend the GUI's functionality and issue commands to TOSSIM. However, because users must pre-compile their plug-ins, this only allows limited interactivity.

Tython (or, Tinython) complements TinyViz's visualization by adding a scripting interface to TOSSIM. Users can interact with a running simulation through TinyViz, a Tython console, or both simultaneously. Tython is based on Jython, a Java implementation of the Python language. These simulation concepts have been discussed in detail in Chapter [2]. This simulation environment is supposed to save time and effort in debugging the sensor applications and also with flexibility to simulate real-world scenarios in the background.

3.2 Architecture

This project involves two kinds of executions – one for the generation of simulation files and the other for running the simulation files, hence two architectures, both essentially three-tier. Figure-3 below, however encapsulates the generic architecture of the project and the specific details are described later.

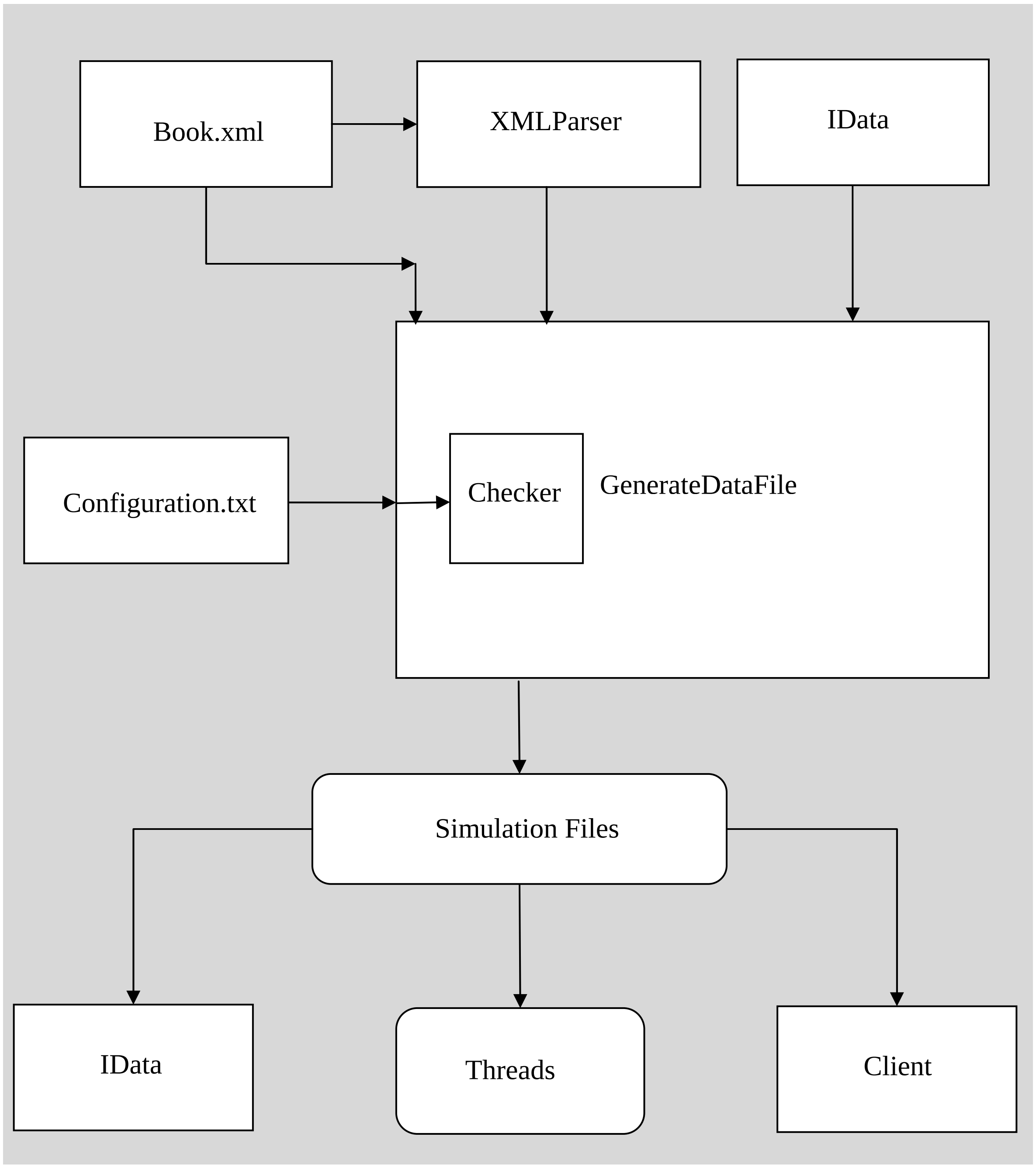

Fig. 3 – Overview of Simulation

3.2.1 Generation Architecture

The components of this part of the project include parsers, socket programs, data and text files. They are used in conjunction with ‘configuration.txt’ and ‘book.xml’ to generate simulation files which can be run later for simulation. The various components that are involved in the generation of simulation files are as below and each of them is explained in detail in Chapter [5] except for one component ‘Checker’ which is discussed in the later part of this chapter, in section 3.3.

- The ‘configuration.txt’ file is first parsed for syntactical errors and correctness by ‘Checker.java’; if it is valid then the same file is given as input to the ‘GenerateDataFile.java’.
- The ‘book.xml’ file that gives the co-ordinates and IDs of the sensors is parsed by ‘XMLParser.java’ (Reference [7]) and objects with these details are created that are used later by other programs.
- The text files ‘addAtBeginningOfIData.txt’ and ‘addAtEndOfIdData.txt’ contain helper functions that need to be copied into the final version of IData.java which contains all the shared data and evaluation functions.
- IData.java contains more helper functions and is used to generate the final version of IData.java
- After ‘configuration.txt’ is verified by ‘Checker.java’, it is parsed by ‘GenerateDataFile.java’ and a ‘SimulationFiles’ directory is created and the files required are either created or copied if they already exist.
- Corresponding to each ‘rule’ in ‘configutation.txt’ file, a java file is created which in-turn corresponds to a thread that encapsulates the constraints expressed in the ‘rule’. Hence, thread files as many as the rules are present.
- The file ‘IData.java’ contains information regarding the boundary values, thresholds, delays, actions and all other data given in the configuration file. This file will be used by all the threads as it contains shared data that is used by these threads.

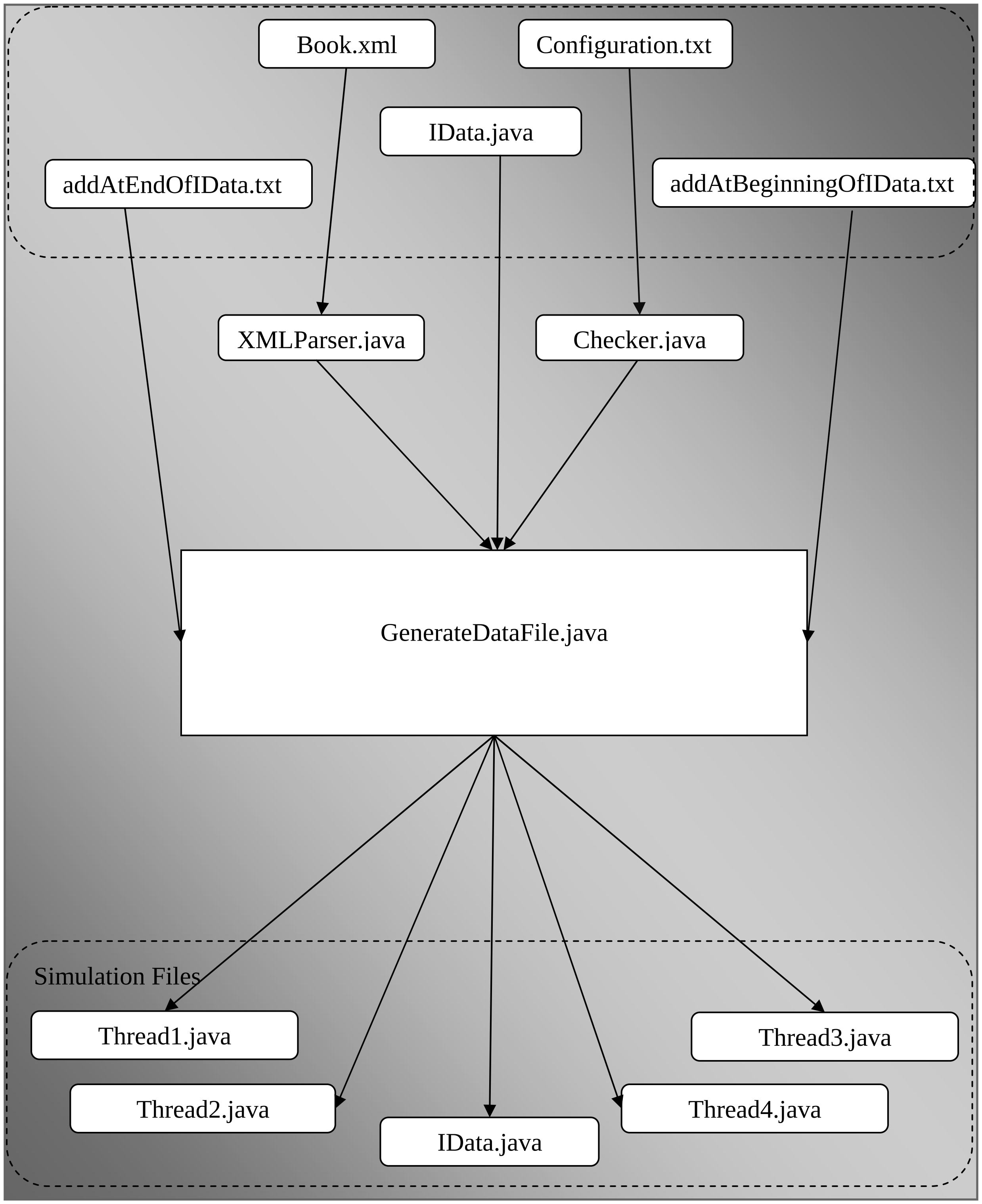

Fig. 4 – Generation Architecture

3.2.2 Execution Architecture

The execution architecture depicts the components, files that are involved during the actual simulation (run-time), the interfaces (communication) between those components and the component's dependencies. The various components of execution architecture are as below:

- 'SenseTask' is an example NesC application, which periodically records the temperature, an ADC value based on which the state of LEDs (red, green, yellow) is decided. This ON/OFF state of the LEDs is used to depict the presence or absence of a condition.
- TinyViz can be launched from console using the command 'TinyViz -run build/pc/main.exe 10'. This launches the TinyViz for the current application, here SenseTask.
- The 'Client.java' file contains code that instantiates each of the thread that has been generated, present in the 'SimulationFiles' directory. Apart from this, it also has a 'CommandOutputReader' object.
- The 'CommandOutputReader' object present in Client.java, after started keeps continuously reading the input stream, where the threads keep writing 'Tython' commands and writes them on the output stream for TinyViz, where these conditions or modifications to the ADC values are applied dynamically.
- IData.java contains the static methods and static data that is shared by all the threads, which has been filled in by the 'GenerateDataFile.java' based on the values present in the 'configuration.txt'. These methods and data values are referenced by the threads during simulation too.
- Each thread, based on the rule to which it corresponds consists of code which has been written based on the 'actions' and 'conditions' which it is uses. These 'actions' and 'conditions' are expanded so that they translate to actual java code. As each thread runs, they keep sending 'Tython' commands to TinyViz to create a scenario, which is the objective of this project.

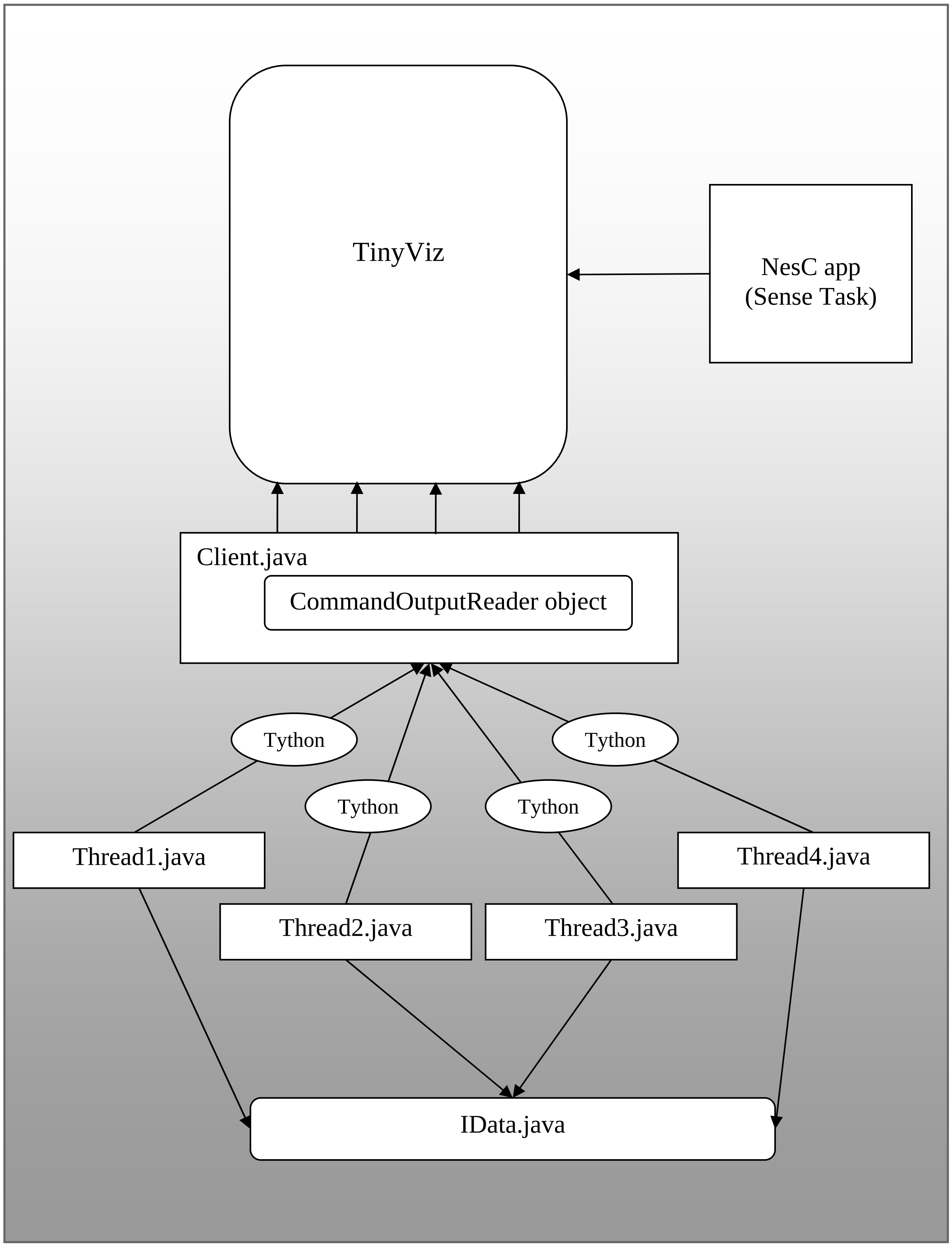

Fig. 5 – Execution Architecture

3.3Verification

The input ‘configuration.txt’ file contains the code that expresses the scenario to be implemented in terms of the grammar, explained in detail, in Chapter [4]. Before this file is actually parsed to generate the simulation files and to read the data, it is verified to check whether the expressions conform to the grammar. This is similar to that of type-checking and lexical-analysis of a compiler to some extent.

3.3.1 Overview of verification

Each tag has a different format and types of data hence, first the tags are checked and methods verify each tag. The java code below that does the job explains this pretty clearly and also indicates the flow of execution. Each tag has different criteria to be checked and hence, a method for each tag. The ‘category’ indicates the type of tag at the current line in the file.

```
switch (category) {
case 1:
        verifyRegion();
        break;
case 2:
        verifyThreshold();
        break;
case 3:
        verifyDirection();
        break;
case 4:
        verifyDelay();
        break;
case 5:
        verifyStartID();
        break;
case 6:
        verifyCondition();
        break;
case 7:
        verifyAction();
        break;
case 8:
        verifyRule();
        break;
case 9:
        verifyOther;
        break;                  }
```

3.3.2 Verification of each block

I. The verifyRegion method checks for three criteria:

- ✓ Proper closing of tags

  (This verification is done for all the tags and the code snippet for this is as below)

  ```
  if (parameter.equals(null) ||parameter.equals("<region>")||
  parameter.equals("<threshold>")||
  parameter.equals("</threshold>")||
  parameter.equals("<direction>")||
  parameter.equals("</direction>")|| parameter.equals("<delay>")||
  parameter.equals("</delay>")|| parameter.equals("<startID>")||
  parameter.equals("</startID>")|| parameter.equals("<condition>")||
  parameter.equals("</condition>")|| parameter.equals("<action>")||
  parameter.equals("</action>")|| parameter.equals("<rule>")||
  parameter.equals("</rule>"))
  {
  System.out.println("<region> tag not enclosed properly");
  System.exit(1);
  }
  ```

- ✓ Formatting of the block (with parenthesis)

  (The code snippet for this is as below)

  ```
  if (!(parameter.startsWith("(") && parameter.endsWith(")"))) {
  System.out.println("<region> block not formatted properly");
  System.exit(1);
  }
  ```

- ✓ Range of Values given (within boundaries of TinyViz UI)

  (The code snippet for this is as below)

  ```
  StringTokenizer st = new StringTokenizer(parameter);
  while (st.hasMoreTokens()) {
  tempStr = st.nextToken("(),");
      if(Integer.parseInt(tempStr)<0||Integer.parseInt(tempStr)> 100) {
      System.out.println("<region> block has out of range values");
      System.exit(1);
      }
  }
  ```

II. The verifyThreshold method checks for three criteria:

- ✓ Proper closing of tag

- ✓ Range of Values given

- ✓ Formatting of the block(with On, Off strings and parenthesis)

  (The code snippet for this is as below)

  ```
  StringTokenizer st = new StringTokenizer(parameter);
  while (st.hasMoreTokens()) {
  ```

```
        tempStr = st.nextToken("(),");
        if (!(tempStr.equals("On") || tempStr.equals("Off"))) {
                if(Integer.parseInt(tempStr)<0||Integer.parseInt(tempStr)
                > 150) {
                        System.out.println("<threshold> block has out of
                        range values");
                        System.exit(1);
                }
        }
    }
```

III. The verifyDirection method checks for two criteria:

- ✓ Proper closing of tag
- ✓ Values given (checks for the validity of the string).

IV. The verifyDelay method checks for three criteria:

- ✓ Proper closing of tag
- ✓ Formatting of the block(with On, Off strings and parenthesis)
- ✓ Range of Values given.

V. The verifyStartID method checks for three criteria:

- ✓ Proper closing of tag
- ✓ Values given. (should be valid to represent IDs of nodes)

VI. The verifyCondition method checks for three criteria:

- ✓ Proper closing of tag
- ✓ Formatting of the block.

VII. The verifyAction method checks for three criteria:

- ✓ Proper closing of tag
- ✓ Formatting of the block.

VIII. The verifyRule method checks for three criteria:

- ✓ Proper closing of tag
- ✓ Formatting of the block.

CHAPTER 4 – Configuration File Grammar

4.1 Overview

Configuration file is a text file that contains the input data by the user, which is enclosed within tags similar to that of an xml document. As of now, there have been eight tags defined in the grammar, but this can be extended if additional features come up. These eight tags correspond to each of the parameters that the user needs to give so as to define a real-time scenario completely. The eight tags that have been used in the current grammar are:

- *region*
- *threshold*
- *direction*
- *delay*
- *startID*
- *condition*
- *action*
- *rule*

The first five tags: “region, threshold, direction, delay, and startID” actually deal with information regarding the physical conditions such as temperature and light. Each of these tags can be used to input data in terms of numerical values (integers or strings) such as boundary values for the given scenario or bounds of a region and thresholds for a condition or direction of propagation.

The last three tags define conditions, actions and rules, which are actually logical constructs of if-then-else statements. The actions and conditions define how the physical values in terms of ADC values of the sensor motes are to be modified or calculated based on specific conditions. Each rule in itself defines a complete scenario, which is also if-then-else statements, but expressed in terms of conditions and actions which have been defined already; typically each rule specifies which condition is to be checked for that scenario and the corresponding action for it.

Each of the above tags has been discussed in detail below, followed by a sample text snippet as to show how the input looks like.

4.2 Region

This tag is used to define the co-ordinates of the region to be identified. The total area on the TinyViz ranges from the co-ordinates (0, 0) to (100,100). Each region is defined by two co-ordinates, the top-left point and the bottom-right point. These two co-ordinates are input by the user to identify each region. The area on the TinyViz screen area might be divided into a finite number of regions, based on the requirement of the scenario or the problem. The input values allowed are integer data types within the specified range (0-100). The values are read as: (top-left x-co-ordinate, top-left y-co-ordinate, bottom-right x-co-ordinate, bottom-right y-co-ordinate) in that order.

```
<region> ::=  (<x co-ordinate1>, <y co-ordinate1>, <x co-ordinate2>,
                <y co-ordinate2>)
```

```
<region>
        (x1, y1, x2, y2)
        (x3, y3, x4, y4)
 </region>
```

4.3 Threshold

Threshold tag is used to define the boundary values or the points beyond which a certain condition is ascertained. There could be lower thresholds and upper thresholds. Hence to define that a condition related to a parameter exists, we define a threshold value related to that parameter, which acts as a boundary value to determine the presence or absence of a particular condition or state. For example, if the pressure beyond a certain value is considered as harmful then that value beyond which the condition is ascertained is defined as the higher threshold. Similarly, if a value of temperature is considered below which there is a potential risk, that value is considered as the lower threshold for that parameter. The values are read as: (On threshold, Off threshold) in that order. The input values allowed are integer data types within the specified range.

On threshold: This defines the upper bound or the upper threshold beyond which a condition respective to that parameter is ascertained. Hence the threshold values with respect to each parameter, i.e, the ADC value are read separately. The values are read as: (OnThresholdForADC0, OnThresholdForADC1, OnThresholdForADC2) in that order.

Off threshold: This defines the lower bound or the lower threshold below which a condition respective to that parameter is ascertained. The values are read as: (OffThresholdForADC0, OffThresholdForADC1, OffThresholdForADC2) in that order.

<threshold> ::= < On threshold> <Off threshold>

<On threshold>::= <OnThresholdForADC0, <OnThresholdForADC1>, <OnThresholdForADC2>

<OffThreshold>::= <OffThresholdADC0>, <OffThresholdADC1>, <OffThresholdADC2>

```
<threshold>
    On(th0, th1, th2)
    Off(th0, th1, th2)
</threshold>
```

4.4 Direction

This tag encloses the information related to the direction in which the given parameter propagates in the scenario to be simulated. Eight of the eight directions are possible, as mentioned below. The input value allowed is a string data type. The values are read as: (direction)

<direction> ::=   <north> | <south> | <east> | <west> | <northeast> | <northwest> | <southeast> | <southwest>

```
<direction>
    north
</direction>
```

4.5 Delay

This tag encloses the information related to delay, which indicates the rate at which the parameter is propagated. The delay implies the rate at which the given scenario propagates or diminishes. This indicates the rate at which the respective parameter needs to be modified in the neighboring areas based on the conditions given. The delay value is given in terms of *milliseconds* for the examples dealt with in the current project. The values are read as: (On delay, Off delay), in that order. The input values allowed are integer data types within the specified range.

On delay: This indicates the rate at which the parameter spreads or the rate at which it affects the parameter, when it is bound to affect an increase in magnitude. This rate is separately noted for each of the parameters. The values are read as: (DelayForADC0, DelayForADC1, DelayForADC2) in that order.

Off delay: This indicates the rate at which the parameter diminishes or the rate at which the parameter spreads when it is bound to affect a decrease in magnitude. This rate is separately noted for each of the parameters. The values are read as: (DelayForADC0, DelayForADC1, DelayForADC2) in that order.

```
<delay> ::= <On delay> <Off delay>
<On delay>::=  <DelayForAdc0> <DelayForAdc1> <DelayForAdc2>
<Off delay>::=  <DelayForAdc0> <DelayForAdc1> <DelayForAdc2>
    <delay>
        On(500,500,500)
        Off(1000,1000,1000)
    </delay>
```

4.6 Start-ID

This tag encloses the information as to where the scenario starts and then propagates from there as per the scenario (expressed as rules later). This might be a series of IDs, representing a group of regions where the scenario is initially present. All the IDs given in the series enclosed in the tags are taken as the starting points of the scenario. The values are read as: (id0, id1…), in that order, though no preference to order is present in this case. The input values allowed are integer data types.

```
<startID> ::= <id0>, <id1>… <IDn> where n>0
<startID>
    Id1, id2
</startID >
```

4.7 Condition

A condition can be given which ascertains the presence of absence of a certain criteria. (For example, a tornado presence at a single point of time may be associated with a high amount of pressure present there attributed with some velocity. Hence, ADC value of pressure may be used to determine this state.) This can be defined based on the parameters and the threshold values that are expressed as ADC values. Each condition needs to be given a *name* so that it can be referenced for use in the rules later on. The conditions need to be of the format of a normal logical construct with some 'Boolean' return value such as an if-else construct. The logical construct is taken as it is into the condition, but the key words such as 'left', the ADC values, the indices, the numerical value, an integer value used for comparison and the equality/inequality symbols used in the expression are recorded for the actual comparisons and calculations.

<condition> ::= <Condition Name> <if condition> <then return value> <else return value>

```
<condition>
    CheckTemp
    if( Left.ADC[0] >= 100) {
    return true;
    else
    return false;
    }
</condition>
<condition>
    CheckExistin
    if( Left.ADC[0] <= 10) {
    return true;
    else
    return false;
    }
</condition>
```

4.8 Action

An action can be defined, that describes which specific parameters are to be modified and set to a specific value, on the regions. Each action corresponds to setting an ADC value that specifies a physical state. (For example, a tornado movement may be simulated by raising and lowering the ADC values that correspond to pressure in regions along the path of propagation which in-turn is given by the 'direction' described above.) Each action needs to be given a *name* so that it can be referenced to be used in the rules later on. The actions need to specify a numerical value of a parameter, which is affectively, an ADC value. This value is noted down and also the index of the ADC value which indicates the ADC value. This is then applied to all the motes present in each region whenever this action is referenced by its name.

```
<action> ::= <Action Name> <apply ADC value>
<action>
    MaxTemp
    ADC[0]=100
</action>
<action>
    MinTemp
    ADC[0]=10
</action>
<action>
    AvgPre
    ADC[1]=50
</action>
```

4.9 Rules

This tag is used to define an independent flow of execution that instructs various actions to be carried out on the regions thereby on motes in them, by checking for various conditions. The actions and conditions are defined already and they are referenced here. Each rule is given a name, so that a thread is created which can be referenced through this name. Then the format follows some logical construct form such as if-else construct. The conditions and actions referenced are replaced by their return values after they are calculated based on the values at run-time. A thread is created for each rule; hence each of the rules is executed independently.

<rule> ::= <rule name> <if Condition Name> <apply Action Name>

```
<rule>
SpreadProcess
if ( CheckTemp == true)
 MaxTemp;
</rule>

<rule>
DistProcess
if ( CheckExistin == false)
 MinTemp;
</rule>
```

CHAPTER 5 – Generation Of Simulation Files

5.1 Overview

The objective of the project is to generate code, programs in java, meaning threads which implement the scenario that has been given by the user in form of the configuration file. The main input for this process is the “Configuration.txt” file based on which the java files are generated, but there are other components needed that are important. These components are required for purposes such as: describing the initial arrangement of the motes, type-checking the configuration file for formatting errors, the parsers required to read the XML formatted files and programs which contain already written helper functions.

However, before the generation of the simulation files, the “configuration.txt” file is checked for correctness of tags and syntax by using the ‘checker’ java file. This plays a similar role to that of a ‘compiler’ before the actual execution, but at a very simpler level. The various parameters checked and how this is performed has already been discussed in detail in section 3.3.

5.2 Inputs and required files

The components involved in generation of simulation files consists of parsers, text files, xml files that contain data, socket programs for data communication and related code. The details of each component that is involved in the generation of simulation files are described in the later part of this section:

- book.xml
- XMLParser.java
- Configuration.txt
- client.java
- CommandOutputReader.java
- GenerateDataFile.java

5.2.1 Book.xml

This contains the initial configuration of the nodes and other details of the nodes such as their initial position in x and y-co ordinates. Also included are the details such as group ID, commands associated, configuration and application associated with. But, in this application only the node ID and the co-ordinates are used.

This component as already mentioned before is existing, (Reference [7]). A sample 'book.xml' file looks like below:

```
<Info>
<Node>
 <Nodeid>0</Nodeid>
 <x>43.56</x>
 <y>52</y>
 <conf>app2</conf>
 <mobility>none</mobility>
 <groupid>0x7D</groupid>
 <radius>100</radius>
 <commands>
    <command>
       <name>turnoff</name>
       <parameters>none</parameters>
    </command>
 </commands>
</Node>
<Node>
 <Nodeid>1</Nodeid>
 <x>48.65</x>
 <y>47</y>
 <conf>app1</conf>
 <mobility>none</mobility>
 <radius>100</radius>
 <groupid>0x7D</groupid>
 <commands>
    <command>
       <name>turnoff</name>
       <parameters>none</parameters>
    </command>
 </commands>
</Node>
</Info>
```

5.2.2 XML Parser

This is an existing software component (References [3]) which reads the data in the above "book.xml" file which parses the details of the configuration and fills the user defined data structures with the attributes of the nodes present in the book.xml file such as the node ID, and its x and y-co ordinates. A node object is created which corresponds to each sensor with all its details such as ID and co-ordinates. The co-ordinates and IDs will be later used in the implementation.

5.2.3 Configuration file

This file contains the user input code which describes the scenario that needs to be simulated. This code follows a grammar which has been defined below under the Section "Grammar for configuration file" in Chapter 4. Various tags have been used to describe different aspects of the scenario. For e.g. the tag <region> is used as an opening tag which encloses the details of each region. </region> is the closing tag which encloses the regions. The tags are used to describe certain parameters. (All these are discussed in chapter 4 in detail):

- the region boundaries
- threshold values which depict presence and absence of various criteria
- direction relevant to the application of the scenario
- delay which indicates the rate at which the conditions are to be propagated
- Starting points- regions for a given scenario.
- sets of conditions expressed in terms of ADC values and logical if structures
- sets of actions expressed in terms of ADC values to be applied
- sets of rules expressed in terms of 'conditions' and 'actions'

This file is kept in the source (src) directory which is then interpreted by the 'GenerateDataFile' program which parses it and fills various data structures and generates new files based on the inputs given in the configuration file. A sample ‘configuration.txt’ file looks like below:

```
<region>
    (0,0,50,50)
    (50,0,100,50)
    (0,50,50,100)
    (50,50,100,100)
</region>
<threshold>
    On(100,90,100)
    Off(40,50,40)
</threshold>
<direction>
    north
</direction>
<delay>
    On(500,500,500)
    Off(1000,1000,1000)
</delay>
<startID>
    1,2
</startID>
<condition>
    CheckTemp
    if( Right.ADC[0]>= 100)
     return true;
    else
    return false;
</condition>
<condition>
    CheckExistin
    if( Right.ADC[0]  <= 10)
     return true;
    else
     return false;
</condition>
<action>
    MaxTemp
    ADC[0]=100
</action>
<action>
    MinTemp
    ADC[0]=10
</action>
```

```
<rule>
    SpreadProcess
    if ( CheckTemp == true)
    MaxTemp ;
</rule>
<rule>
    DistProcess
    if ( CheckExistin == false)
     MinTemp ;
</rule>
```

5.2.4 Client Program

This is the program which manages the sockets for connection between the applications written in NesC code and the java programs which contain the 'Tython' commands that are responsible for applying the conditions described for the scenario in the configuration file. This program opens the sockets, creates all the thread objects and starts them from the files that have been generated. A sample code snippet from the 'client.java' program is as below:

```
// Initialize the IData.java which contains static data and methods
IData.initializeVar();

// Create ports for communication
smtpSocket = new Socket("localhost", 9999);
os = new DataOutputStream(smtpSocket.getOutputStream());
is = new DataInputStream(smtpSocket.getInputStream());
is = new DataInputStream(System.in);

// Create the CommandOutputReader object and start it
CommandOutputReader cmdReader = new CommandOutputReader(is);
cmdReader.start();

// Write the initialization commands to start communication
os.writeBytes("from simcore import *\n");
os.writeBytes("sim.resume()\n");
sleep(5000);
os.writeBytes("sim.pause()\n");

// Start the thread objects from the generated files
// starting the thread
SpreadProcessThread spth = new SpreadProcessThread();
spth.start();
// starting the thread
```

```
DistProcessThread exth = new DistProcessThread();
exth.start();

// reads the input stream from the standard input
while (true) {
        String line = is.readLine();
        os.writeBytes(line + "\n");
}
```

5.2.5 CommandOutputReader

This program is responsible for reading the ‘Tython’ commands that are written continuously on the input stream, by the threads that implement the rules given in the configuration. The threads based on the conditions, implement various actions continuously issuing ‘Tython’ commands to TinyViz. The commands are read and sent to the TinyViz where the conditions are applied. A CommandOutputReader object is created by the client.java program and started, to manage the communication. The code snippet that summarizes the functionality of this class is:

```
while(true)
     {
        String line=os.readLine();
        System.out.println(line+"\n");
     }
```

5.2.6 GenerateDataFile

This is the component that is responsible for the actual parsing of the configuration text file. A separate directory to place all the generated simulation files is created and various source files required for simulation are copied into it. These include:

- book.xml
- XMLParser.java
- CommandOutputReader.java
- configuration.txt
- files “addAtBeginningOfIData.txt” & “addAtEndOfIData.txt” which contribute to IData.java
- client.java

Copy required files	Create folder for simulation files
	Copy book.xml
	Copy client.java
	Copy CommandOutputReader.java
	Copy XMLParser.java
	Copy addAtBeginningOfIData.txt
Parse Configuration and write to IData	Parse regions
	Parse threshold values
	Parse direction
	Parse delay
	Parse startID
	Parse conditions
	Parse actions
	Parse rules
Copy required files	Copy addAtEndOfIData.txt

Fig. 6 – GenerateDataFile Structure

Apart from copying the required files, the configuration file is parsed as per the grammar defined and all the data and conditions are extracted and used to generate the simulation files. All the extracted data is written to one “IData.java” file, which contains this as static data. The pseudo code for the GenerateDataFile is as below:

```
create SimulationFiles folder
    copy book.xml
    copy client.java
    copy CommandOutpurReader.java
    copy XMLParser.java
    open IData.java to write static data and methods
            copy to IData.java few already existing helper functions
            from addAtBeginningOfIData.txt
    open configuration.txt file

while (! End of File) {
    check for <region> tag
            read north-east x-coordinate
            read north-east y-coordinate
            read south-west x-coordinate
            read south-west y-coordinate
            write to IData
    if </region> found, exit this block

    check for <threshold> tag
            read On threshold for ADC[0]
            read On threshold for ADC[1]
            read On threshold for ADC[2]
            read Off threshold for ADC[0]
            read Off threshold for ADC[1]
            read Off threshold for ADC[2]
            write to IData
    if </ threshold> tag found, exit this block

check for <direction> tag
            read 'direction' string
            write to IData
if </direction> tag found, exit this block

check for <delay> tag
            read On delay for ADC[0]
            read On delay for ADC[1]
            read On delay for ADC[2]
            read Off delay for ADC[0]
            read Off delay for ADC[1]
            read Off delay for ADC[2]
            write to IData
if </delay> tag found, exit this block

check for <startID> tag
            read series of startIDs
            write to IData
if </ startID > tag found, exit this block

check for <condition> tag
            read name of Condition
            read strings such as left/right/upper/lower
            read which parameter to consider, ADC value index
            read the parameter value, magnitude
            read the relation symbol, '<' or '>'
            write to IData
if </condition> tag found, exit this block
```

```
check for <action> tag
        read name of the action
        read ADCValue index to know which parameter
        read magnitude of target value
        write to IData
if </action> tag found, exit this block

check for <rule> tag
        read name of rule
        create a thread file with this rule name
        read which condition this rule uses
        read which action this rule implements
        write to IData
if </rule> tag found, exit this block

    copy to IData.java few already existing helper functions
    from addAtEndOfIData.txt

} end while
close all open files and newly created files.
end
```

The code in this file reads, interprets the input given in the configuration file, initializes IData.java file and also creates simulation files based on the input configuration. Various files are created and written, each of which implements a thread for each separate scenario or rule, based on the rules given in the configuration file.

5.3 Parsing the Configuration file

Configuration file is the input given by the developer (user in this case) in which constraints and conditions of the environment are given, which have to be implemented and run in the background to test the sensor applications, so as to simulate the original environmental conditions. The details of the configuration file and the grammar in it are discussed in chapter [4].

The file is parsed using string parsing operations in java. Each of the pre-defined strings such as 'region', 'threshold', 'direction', 'delay', 'startID', 'condition', 'action' and 'rule' are searched, parsed and recorded by "GenerateDataFile.java".

5.4 Construct threads for each Rule

The “GenerateDataFile.java” mentioned above also notes the details from the “configuration.txt” file and writes them to “IData.java” which contains all the structures, details and helper, initialization functions needed by the threads that actually implement the constraints.

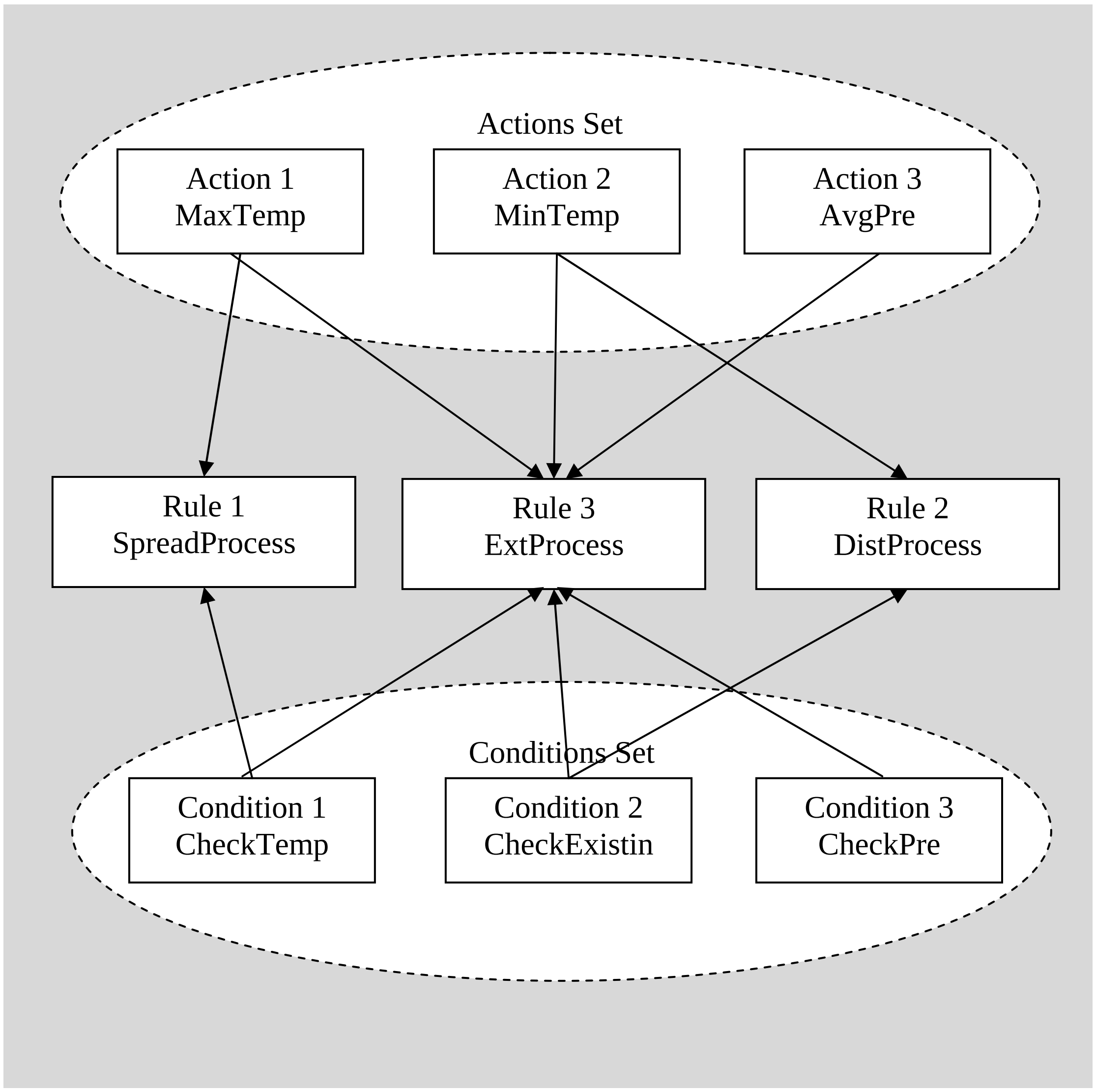

Fig. 7 – Actions, Conditions and Rules

The threads are defined by the ‘rules’ in the configuration file that are in-turn expressed in terms of ‘actions’ and ‘conditions’. The actions define the setting that needs to be eventually implemented on the sensor motes in a specific region, which is the ADC value setting. The conditions define the pre-requisites for any action to be implemented. The rules make use of these two expressions and define constraints, in form of conditions and the ADC value settings using the actions. Figure-3 represents how the rules are made of actions and conditions and explains that each rule is made of combination of one or more actions and conditions.\

5.5 Helper Functions

There are various functions used in the generation of simulation files, to find the neighboring regions, modify ADC values of all sensors in a particular region and to evaluate functions based on the parameters given. These are mostly present in “addAtBeginningOfIData.txt” and “addAtEndOfIData.txt”. Below are few helper methods that are used in ‘IData.java” file.

```
❖ public static int returnArrayValue(int i, int j) {
          return Region[i][j];
  }
❖ public static int getLocationID(int x1, int y1) {
          if (x1 >= 0 && y1 >= 0 && x1 <= 100 && y1 <= 100) {
                  for (int ii = 0; ii < IData.numAreas; ii++) {
                  if (x1 >= IData.Region[ii][0] && x1 < IData.Region[ii][2]&& y1
                  >= IData.Region[ii][1]&& y1 < IData.Region[ii][3]) {
                  return ii;
                  }
                  }
          }
          return -1;
  }
❖ public static int getLeftID(int id) {
          if (IData.Region[id][0] - 1 > 0) {
                  // check the region just below at the starting border
                  int tempIDs = IData.getLocationID( IData.Region[id][0] - 1,
                  IData.Region[id][1] + 1);
                  return tempIDs;
                  }
          else
                  return -1;
  }
❖ public static int getRightID (int id) {
        if (IData.Region[id][2] + 1 < 100) {
                // check the region just below at the starting border
                int tempIDs = IData.getLocationID(IData.Region[id][2] +
                1,IData.Region[id][1] + 1);
                return tempIDs;
        }
        else
                return -1;
}
❖ public static int getLowerID (int id) {
        if (IData.Region[id][3] + 1 < 100) {
                // check region just below @ the starting border
                int tempIDs = IData.getLocationID(IData.Region[id][0] + 1,
                IData.Region[id][3] + 1);
                return tempIDs;
        }
        else
                return -1;
}
❖ public static int getUpperID (int id) {
        if (IData.Region[id][0] - 1 > 0) {
                // check the region just above at the starting border
                int tempIDs = IData.getLocationID(IData.Region[id][0] +
                1,IData.Region[id][1] - 1);
                return tempIDs;
        }
        else
                return -1;
}
❖ public static boolean validateFunction (String direction, int adc, String
comparator, int threshold) {
        int affectingID = -1;
        if (direction.equals("Left"))
                affectingID = getLeftID(id);
        else if (direction.equals("Right"))
                affectingID = getRightID(id);
        else if (direction.equals("Lower"))
                affectingID = getLowerID(id);
        else if (direction.equals("Upper"))
                affectingID = getUpperID(id);
        while (affectingID != -1) {
                if (comparator.equals(">=")) {
                        if (IData.detected[affectingID][adc] >= threshold)
                                return true;
                        else
                                return false;
                } else if (comparator.equals("==")) {
                        if (IData.detected[affectingID][adc] == threshold)
                                return true;
                        else
                                return false;
                } else if (comparator.equals("<=")) {
                        if (IData.detected[affectingID][adc] <= threshold)
                                return true;
                        else
                                return false;
                } else if (comparator.equals("<")) {
                        if (IData.detected[affectingID][adc] < threshold)
                                return true;
                        else
                                return false;
                } else if (comparator.equals(">")) {
                        if (IData.detected[affectingID][adc] > threshold)
                                return true;
                        else
                                return false;
                } else {
                        System.out.println("input format error \n");
                        return false;
                }
        }
        return false;
}
```

CHAPTER 6 – Example

6.1 Fire Spreading

One very relevant area of deployment of sensors that may be considered is that of a vast spread of area, representing a farm or a forest. Java threads need to be generated for this scenario, which simulate the temperature rise, with an increase in the ADC values of the motes in the regions.

As already discussed, the area displayed by TinyViz is divided into regions, as defined by the user. There is random distribution of sensors in these regions which are specified by ‘book.xml’ file. This file is parsed and the IDs of motes associated with each region are noted. The direction input indicates which way the temperature rise proceeds. This may be viewed by using the states of the LEDs on sensors, consider red represents temperature rise in the region of the sensor. As fire extends an area, the motes in that region all get an ADC value (here 100), representing the physical state of the sensor.

6.1.1 Algorithm for spreading

Each region is identified as an entity which decides periodically the states of the motes which fall under it. As an example, if the direction is towards “north”, meaning upwards, as the fire spreads upwards, the region always checks if the region below has the temperature enough to represent fire. If it happens, then the current region also changes the ADC values of all the sensors under it to represent high temperate values, (here 100) to indicate the fire spreading. In the next round of polling, this region becomes able to spread it to the next region in line. This process continues till the end.

After the spreading thread has finished the last region, since in reality fire ultimately extinguishes, so after a delay, the extinguishing thread starts and spreads in the same direction as that of the fire, but this time setting the ADC values to a value (here 0 or 10) that represents a state where there is no fire. This algorithm is well represented in the figure-5, where nine regions are considered and the spreading and extinguishing are represented.

The pseudo-code for the algorithm is given below. Here, the condition and action are already given as per the rule in the configuration file. These conditions and actions are the building blocks that define what ADC values needs to be checked and what the final value should be set. These are referred as given in the pseudo-code below. The GenerateDataFile records the definitions of the actions and conditions similar to that of macros and when they are referred by the rules, they are replaced by their original definitions, where the truth values are returned. The same algorithm applies to the extinguishing thread. At the end of this algorithm, each time it is checked if all the regions have reached the destination state, if so the second thread is started. This may also be done, by specifically starting the second thread which then sets the ADC values to reflect the new state.

```
For each region:
        delay;
        check ADC value of adjacent region, based on direction
        if (current region doesn't satisfy final condition)
          if (adjacent region is capable of spreading)
              for each sensor in the region:
                assign new ADC value
          end if
        else if (current region satisfies final condition)
          if (current region is not capable of spreading)
                set a flag to indicate current region is capable of spreading
          end if
        end if
end region;
```

Fig. 8 – Algorithm for spreading

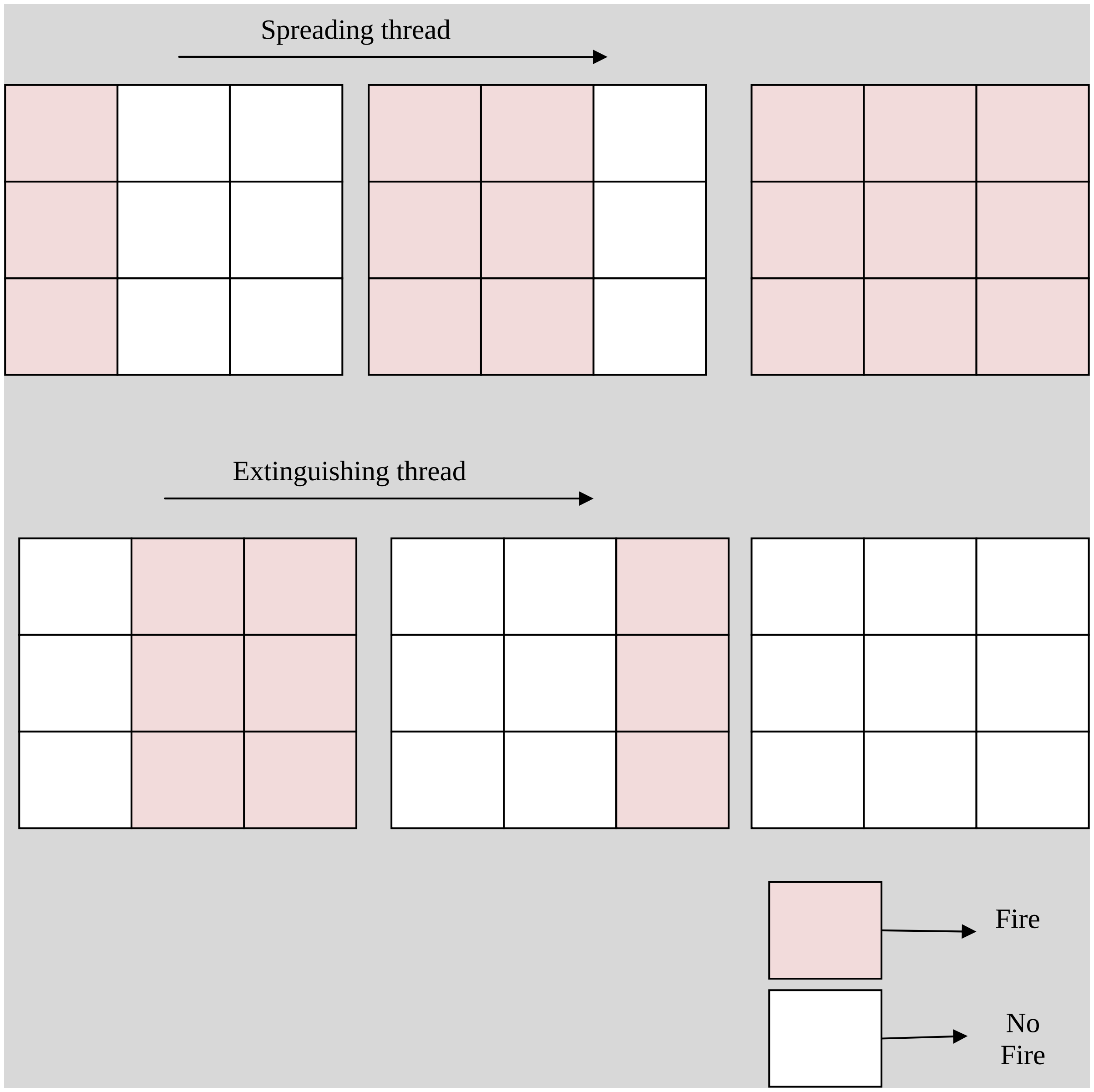

Fig. 9 – Visualization of change in ADC values by concurrent threads

6.2 Conclusion

In many applications, based on TinyOS and NesC, there is more application to this plug-in. Examples such as diffusion of gases, object movement tracking, photo-sensing, tornados and other such environmental situations might be converted into problems which can be represented as variations in physical parameters such as temperature and pressure. These problems can make use of this plug-in, by expressing the conditions in terms of the ADC values that are understood by the sensors.

CHAPTER 7 – Eclipse Plug-in and Testing

7.1 Introduction

The project has been converted as an eclipse-plug in for portability and ease of use for testing NesC applications, before they are deployed.

Eclipse is an open platform. It is designed to be easily and infinitely extensible by third parties. At the core is the eclipse SDK, around which various tools can be built. These products or tools can further be extended by other products/tools and so on. The extensibility of Eclipse is achieved by creating products/tools in form of plug-ins.

The actual program and required files are converted in ".jar" files after which a plug-in project is used to convert them into a plug-in. The current plug-in, though has a lot of data required in form of many files, most of them are common for all the scenarios; only the configuration file given by the user varies. Hence, a new project type is created, which accepts the configuration file and gives it as input to the plug-in, which using a command from the customized menu, is used to generate the simulation files, in a separate directory.

7.2 Testing

7.2.1 Input Values

The input given by the user, based on the grammar is tested for validity. Most of the tests include well-formed-ness, validity of the input values and strings, for proper syntax.

- Input Value: In Configuration.txt, region tag "(0,0,50,50" entry
  Result: <region> block not formatted properly
- Input value: In region tag "(0,50,50,101)" entry
  Result: <region> block has out of range values
- Input value: In region tag "(0,50,50,-10)" entry
  Result: <region> block has out of range values
- Input value: In threshold tag "On(100,90,100) (40,50,40)" entry

Result: Region verified

<threshold> block not formatted properly

- Input value: In threshold tag “(100,90,100) Off(40,50,40)” entry

  Result: Region verified

  <threshold> block not formatted properly

- Input value: In threshold tag “On(100,90,100 Off(40,50,40)” entry

  Result: Region verified

  <threshold> block not formatted properly

- Input value: In direction tag “northe” entry

  Result: Region verified

  threshold verified

  <direction> direction name invalid

- Input value: In delay tag “On(500,500,500” entry

  Result: Region verified

  threshold verified

  direction verified

  <delay> block not formatted properly

- Input value: In startID tag “1,2,200” entry

  Result: Region verified

  threshold verified

  direction verified

  delay verified

  <startID> block has out of range values

- Input value: In condition tag “<rule>” entry

```
<condition>
CheckTempHigh
if( Right.ADC[0]>= 100)
 return true;
else
 return false;
<rule>
</condition>
```

  Result: Region verified

  threshold verified

direction verified

delay verified

startID verified

<condition> block has out of range values

7.2.2 Snapshots

To invoke the TinyViz GUI and associate with a NesC program, The command below can be used.

#tinyviz –run build/pc/main 10

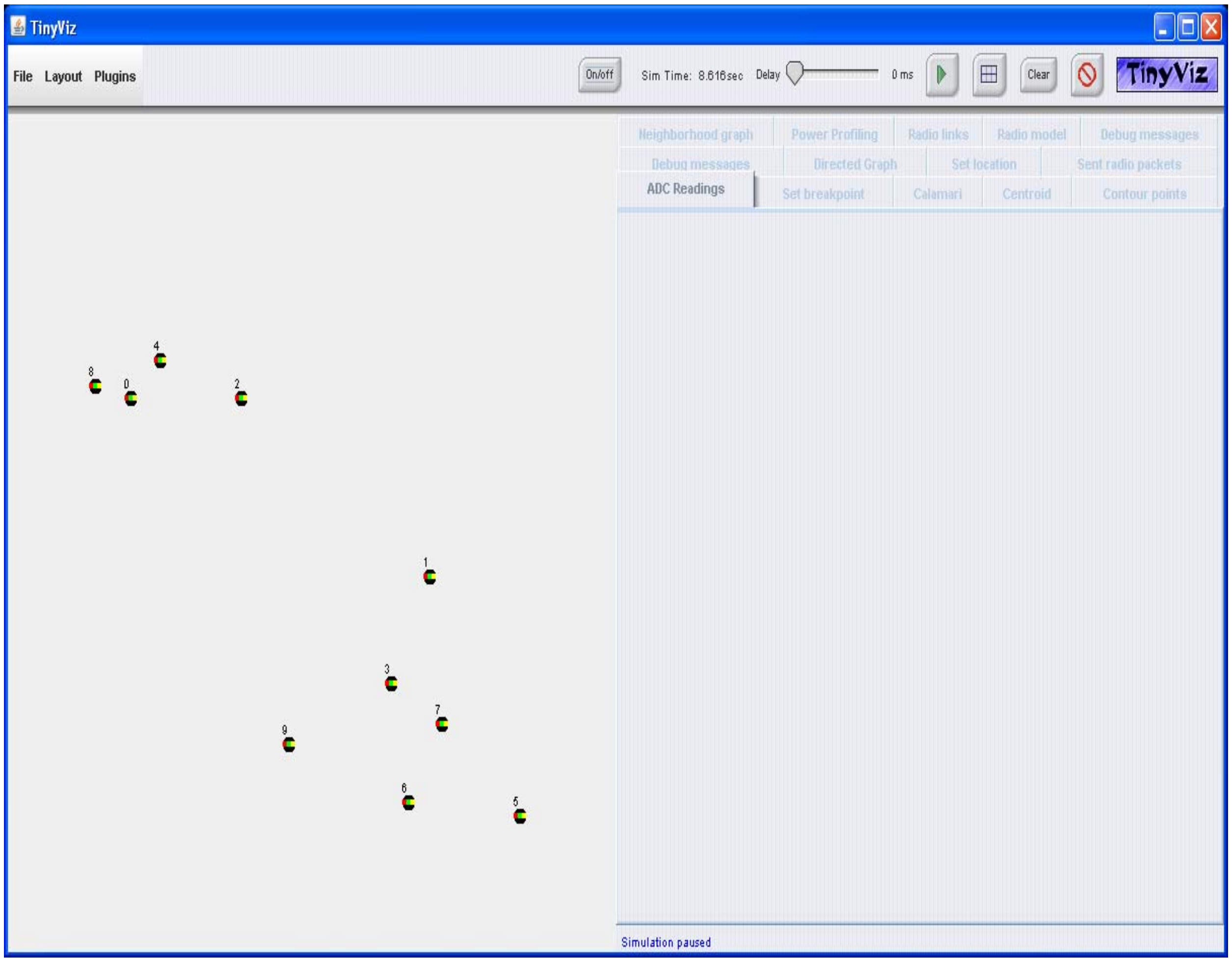

Fig. 10 – TinyViz GUI initially for 10 motes

Figures-10 show snapshots of TinyViz, with 10 motes before the simulation of fire-spreading with the left part of the window showing the simulation, the right part of the window is for the plug-ins for TinyViz. Plug-ins can be activated or de-activated using the File menus. Simulation can be paused, resumed, stopped using the options on the top-right corner of the window.

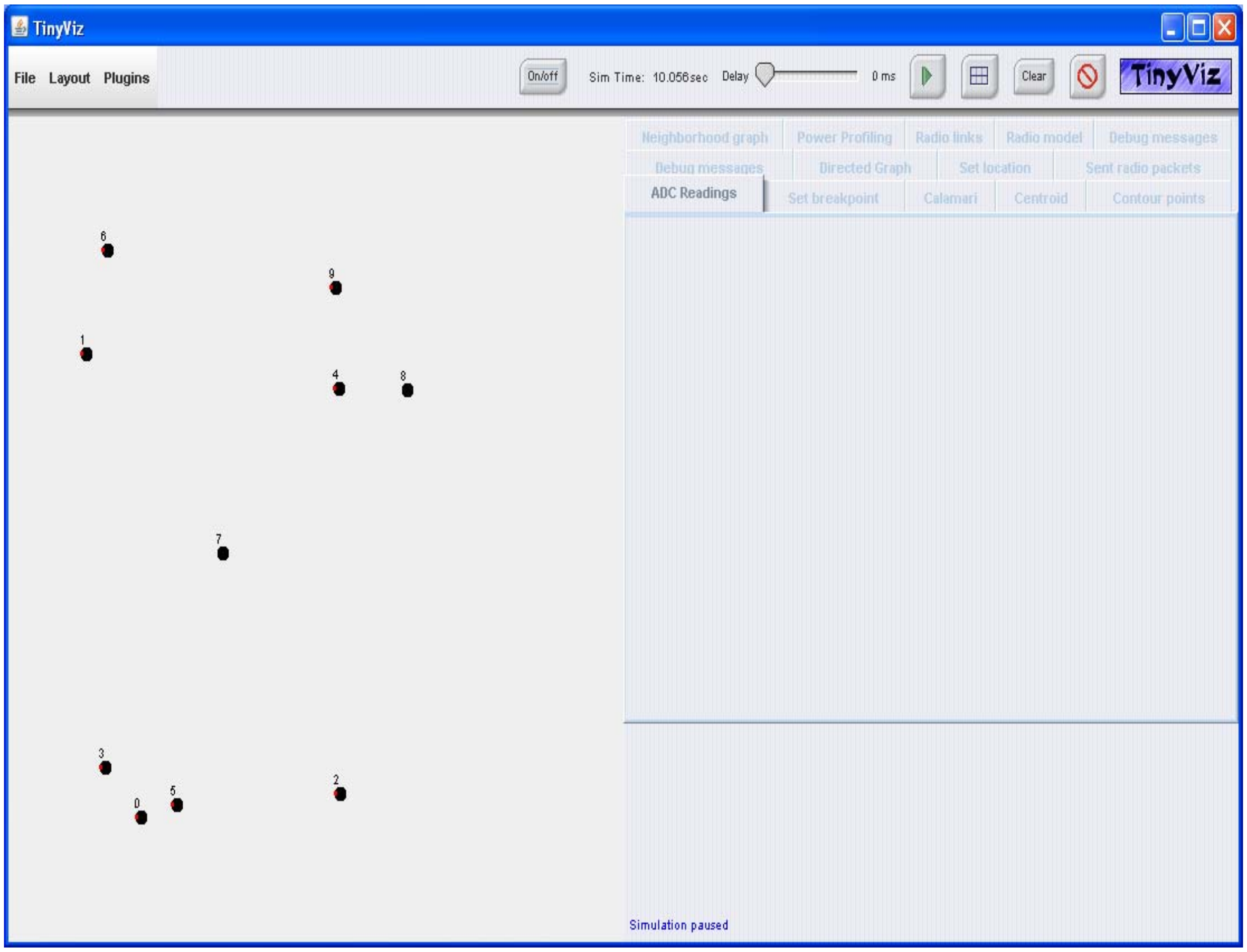

Fig. 11 – TinyViz GUI after threads run for certain period

7.2.3 Performance and limit

As the number of rules increases and thereby the numbers of threads, the commands to the sensors/motes are more delayed and if there are more than 10 threads, at least one sensor is not being able to receive any message at all. As this number increases, more and more motes stop receiving commands due to system processor not being able to schedule all the threads as expected.

CHAPTER 8 – Conclusion and Extensions

8.1 Conclusion

The objective of this project was to develop a plug-in that simulates real-world scenarios to be simulated on TinyViz. The current plug-in takes the user input of constraints in the format of a configuration file. Many conditions in the environment may be simulated using this plug-in if each of those states may be represented as conditions in terms of the ADC values which are understood by the sensors. Then, they can be simulated in TinyViz, on which the NesC application can be tested.

8.2 Future Work

- As the field of sensors networks grows, there might be more and more areas where sensors are used, and hence, new expressions and new ways to represent physical conditions may be identified and incorporated with the current plug-in.
- The current plug-in takes as input the configuration and creates threads; however this might not be applicable to all kinds of problems. Hence, it might be extended to have more types of inputs in other forms.
- A GUI may be developed for users to provide the information. This helps in reducing the user time in following a specific grammar and writing a configuration.
- For large-scale applications, creating threads more than a specific number might not be a good idea, hence new algorithms are needed, which reduce the number of threads by grouping similar tasks into a single thread. This plug-in creates a thread for each rule and hence, this aspect may be improved to use with huge applications.

References

[1] TinyOS

http://www.tinyos.net/tinyos-1.x/doc/tutorial/index.html

[2] TinyOS documentation

http://docs.tinyos.net/index.php/TinyOS_Documentation_Wiki

[3] TinyOS Wiki

http://en.wikipedia.org/wiki/TinyOS

[4] NesC

http://nescc.sourceforge.net/

[5] Tython

http://www.tinyos.net/tinyos-1.x/doc/tython/manual.html

[6] Tython by Mike Demmer and Phil Lewis

http://www.cs.berkeley.edu/~demmer/talks/tython-nest-04.ppt

[7] Representation of mote settings in XML and parsing by Sandeep Pulluri

krex.k-state.edu/dspace/bitstream/2097/1012/1/SandeepPulluri2008.pdf

[8] Eclipse plug-in development

http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html

[9] Plug-in Development Environment Overview

http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.platform.doc.isv

[10] Eclipse Plug-in

http://www.eclipsepluginsite.com/index.html