CEP ENGINE FOR EMBEDDED DEVICES: A POC USING SIDDHI

Sritharan Nadeshkumar

(118221C)

Degree of Master of Science

Department of Computer Science & Engineering

University of Moratuwa

Sri Lanka

September 2016

CEP ENGINE FOR EMBEDDED DEVICES: A POC USING SIDDHI

Sritharan Nadeshkumar

(118221C)

Thesis submitted in partial fulfillment of the requirements for the degree Master of Science in Computer Science

Department of Computer Science & Engineering

University of Moratuwa

Sri Lanka

September 2016

DECLARATION OF THE CANDIDATE AND SUPERVISOR

I declare that this is my own work and this thesis does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my thesis, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

…… …….
S. Nadeshkumar Date

The above candidate has carried out research for the Masters thesis under my supervision.

…… …….
Dr. H. M. N. Dilum Bandara Date

ABSTRACT

With the advent of Internet of Things (IoT), sensors are placed everywhere. These sensors generate continuous streams of data that need to be processed in near real time. Complex Event Processing (CEP) was introduced to achieve this requirement. Currently, all data generated from the sensors are directed to CEP engine(s) in servers or cloud backend to take decisions.

Aim of this research is to push the CEP capabilities towards the sensors, actuators, and gateways nodes (i.e., embedded devices that exist adjacent to these sensors). This enables many decisions to be made locally while reducing the response time. Moreover, this could substantially reduce the volume of data transmitted through the network to traditional CEP engines considerably freeing up the network bandwidth. Moreover, it reduces the computational requirements and cost of servers/cloud.

We develop a CEP engine for resource constrained embedded devices to be placed at the edge of the IoT network. The proposed CEP engine is developed for Arduino, as it is a globally popular, open source hardware platform with a massive user base. In addition, it uses Siddhi Query Language, which is similar to SQL queries. Proposed CEP engine adopts a single threaded model because majority of the embedded devices including Arduino are single threaded. Proposed CEP engine was designed for predefined queries, over dynamic query assignment, as embedded devices have limited memory and CPU resources. Proposed CEP engine uses a state machine to implement the Pattern and Sequence type queries, and uses tuple-type data structure for internal processing. It supports Pass through, Filter, Window, and Join type queries as well. Utility of the proposed CEP engine is demonstrated using sensor and actuator system developed using an Arduino UNO board. Performance analysis demonstrated that the proposed CEP engine is capable of handling over 300 simple filter queries per second in Arduino UNO.

Keywords: Complex Event Processing, Embedded Device, Event Processing, Internet of Things, Siddhi.

ACKNOWLEDGEMENTS

I would like to express sincere gratitude to my supervisor, Dr. Dilum Bandara, for his invaluable support, encouragement, and continues guidance throughout the project. His continuous guidance enabled me to complete my work successfully and within the timeline.

I am indebted to Dr. Srinath Perera for his valuable guidance and the feedback, which helped me to complete the work successfully.

I would like to thank the department of computer science, University of Moratuwa to giving me this opportunity and continuous support throughout my MSc course.

I am grateful for the support and assistance provided by the management of IronOne technology and colleagues at work, who supported throughout my MSc course. Specially, I like to mention Marko Garafulic, Buddhika Jayasinghe, and Pubudu Maggonage for their tremendous support to ease my workload on IronOne technology that gave me peace of mind to focus on my studies.

It is a pleasure to thank my colleagues in the University for providing friendly and a cheerful environment.

I am especially indebted to my parents for their love, encouragement, and support throughout my life. I thank my wife Abarna, for always being my joy and my guiding light, and my sweet little sons Abinash and Nikaash, for making those joyful moments in home and being very understandably, allowing me to study when required.

I like to extend my grateful to Mr. Sachithanandan for recommending me for the MSc, and encouraging me to complete the course.

Finally, I thank all who supported me in every respect during the MSc, for this project would not have been possible without their support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ANTLR	Another Tool for Language Recognition
API	Application Program Interface
BNF	Backus-Naur Form
CEED	CEP engine for embedded devices
CEP	Complex Event Processing
CPU	Central Processing Unit
DFS	Depth First Search
DSDM	Dynamic Systems Development Method
EPL	Event Processing Language
FOSH	Free and Open Source Hardware
FPGA	Field-Programmable Gate Array
GPL	General Public License
GUI	Graphical User Interface
IC	Integrated Circuit
IDE	Integrated Development Environment
IO	Input/Output
IoT	Internet of Things
IP	Internet Protocol
ISV	Independent Software Vendor
ITU	International Telecommunication Union
JVM	Java Virtual Machine
LCD	Liquid Crystal Display

LED	Light Emitting Diode
LiSEP	Lightweight Stage-based Event Processor
OEM	Original Equipment Manufacturer
PC	Personal Computer
POC	Proof of Concept
POJO	Plain Old Java Object
PWM	Pulse Width Modulation
QoS	Quality of Service
RAM	Random Access Memory
SEDA	Staged Event Driven Architecture
SQL	Structured Query Language
USB	Universal Serial Bus
XML	EXtensible Markup Language

Chapter 1

INTRODUCTION

1.1 Background

Explosive growth of smart phones and tablet PCs have substantially increased the number of devices connected to the Internet. In 2008 the number of devices connected the Internet surpassed the total population [1]. While the number of devices continue to increase, also their computing, storage, and communication capabilities are on the rise. Alternatively, with the advances in sensor technology, sensors are becoming more powerful, cheaper, and smaller. This has led to the proliferation of embedded, smart, and computationally rich devices with multitude of sensors. These technological changes and large-scale adoption are the main contributors for the emergence of the Internet of Things (IoT) concept.

With the popularity of the IoT, sensors are been placed everywhere. These sensors generate, continuous streams of data, and in many cases, those streams need to be processed in near real time. To process these continuous streams in real time, a new form of data processing called Complex Event Processing (CEP) was introduced. Real-time stock trading, surveillance, and event monitoring are some of the popular CEP applications. The common requirement among these applications is that they need to be able to continuously collect, process, and analyze data in real time, as well as produce results immediately, even when data arrive at very high rates.

These requirements cannot be handled by the relational databases, as they are designed to collect data and store it continuously. Here, you can subsequently analyze and filter those data by manually supplying queries to the system. Conceptually CEP is an inverted version of the traditional database, as it stores query in the system and run continuously on the incoming data.

As millions of sensors generate continuous data, it is possible to imagine the magnitude of data to be transferred to the CEP engines, which is usually placed some distance away from the actual sensors. This requires upgrade to network systems and technologies to be in par with the expensive network traffic requirements. Figure 1.1 illustrates the current state of the IoT. Currently, all data generated from the sensors need to be directed to servers or a cloud backend. The backend runs CEP engine to take decisions, and communicate those decisions back to sensors and actuators to perform various actions.

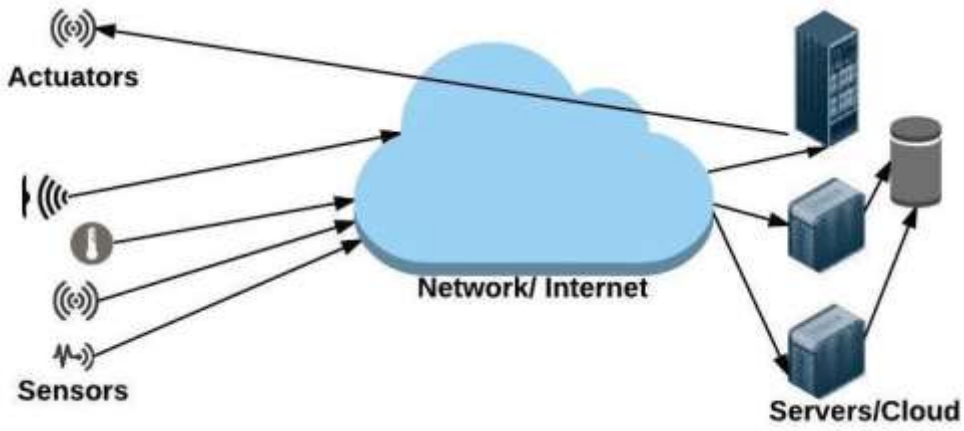

Figure 1.1: Current state of IoT with CEP engines in powerful servers/cloud.

Industrial process automation systems are adopting event-based communication. There is a growing tendency to push the control down towards the hardware, so that the decision can be taken at low level, and the communication bandwidth (and energy for that too) can be spared. This saves the cost for upgrading network systems. As a consequence of pushing the control functionality downwards, lightweight embedded devices should be able to recognize and react to events [2]. Therefore, as illustrated in Figure 1.2, CEP applications often need to be run on embedded devices to quickly react to detected events (within the device) while significantly cutting down the number of less interesting events that is pushed upstream. This not only increases the responsiveness of a particular sensor and actuator application but also reduces the upstream bandwidth requirements and energy consumption.

In addition, several use cases require the local decision, for example say the fire detection system employed for a building and respective CEP engine deployed in cloud. In case of fire, there could be a chance that the connectivity could have disconnected before the fire detected by CEP, and in this situation having local CEP will only serve the purpose.

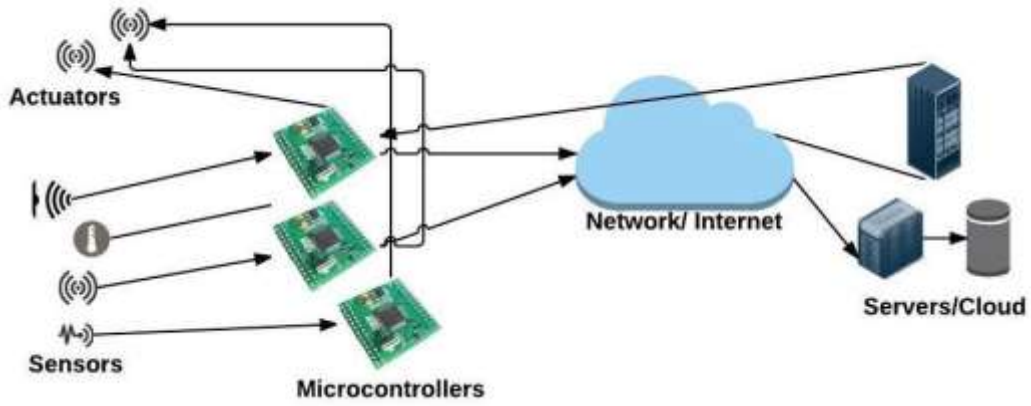

Figure 1.2: Ideal state of IoT with CEP engines in micro controllers near sensors and actuators.

1.2 Problem Statement

The main goal of this project is to develop a Complex Event Processing engine that runs on resource-constrained embedded devices. Our main objective is to build an open source, CEP Engine that runs on Arduino-based embedded devices while supporting the Siddhi Query language.

We selected Arduino-based embedded devices, as they a globally popular, open source hardware platform with a massive community base. In addition, our CEP engine for embedded devices uses Siddhi Query Language, which is similar to SQL queries. This enable rapid development of IoT applications as the CEP capabilities can be added to embedded devices just by writing an SQL-like query.

Key benefits of this design are as follows:

- As the proposed CEP engine to be deploy in embedded devices, CEP engine can sit very close to the actual sensors. Thus, the filtering occurs at data

source (i.e., first mile), greatly reducing the volume of the data transferred through the network to the back end.

- CEP engine based on open hardware like Arduino provides the flexibility of easily modifying the hardware for actual needs by users.
- As the proposed CEP engine supports several popular CEP features like filter, pattern matching, and sequence, it can be easily used as a standalone application in small deployments.
- Application configuration and uploading the program to embedded device are through the website and automated script enables anyone to use the proposed CEP engine.

1.3 Outline

The reminder of the thesis is organizes as follows. Chapter 2 presents the background of this research, which covers IoT, CEP systems, and edged hardware. Chapter 3 presents the proposed architecture of the CEP Engine for embedded devices. Design constraints are also discussed. Chapter 4 discusses the methods, tools, standards and approaches used in implementing the proposed CEP Engine. Chapter 5 presents the performance evaluation. Chapter 6 summarizes limitations, known issues, conclusion, and suggests future works.

Chapter 2

LITERATURE REVIEW

Related work on Internet of Things (IoT), Complex Event Processing (CEP) systems, and embedded devices are discussed, as CEP engine for embedded devices embraces those domains. Siddhi architecture is discussed in detail while surveying on other popular open source CEP engines. In addition, we have looked into the open hardware microcontroller implementations while Arduino is discussed extensively.

2.1 Internet of Things

2.1.1 Background

The initial idea of Internet of Things (IoT) that every items were connected to the Internet by sensor devices such as RFID (Radio Frequency Identification) to accomplish intelligent recognition and network management, was first proposed in 1992 [3]. Wireless sensor networks and RFID are the core support technologies for IoT. The concept of IoT was addressed in International Telecommunication Union (ITU) Internet reports 2005[3], where it reported that everything could be connected with each other at any place and in any time by utilizing technologies such as RFID, wireless sensor networks, intelligent embedded devices, and nanotechnology [3].

Due to lack of a standard definition for Internet of Things, and every paper defined its own definition, it seems the following definition addressed the entire vision of Internet of things as illustrated in Figure 2.1.

"The Internet of Things allows people and things to be connected Anytime, Anyplace, with Anything and Anyone, ideally using Any path/network and Any service." [4]

Figure 2.1: Definition of Internet of Things (IoT) [4].

As shown in Figure 2.1, the above definition for IoT implies, addressing elements such as Convergence, Content, Collection (Repositories), Computing, Communication, and Connectivity in the context where there is seamless interconnection between people and things and/or between things and things [4]. The IoT implies a symbiotic interaction among the real/physical, the digital/virtual worlds: physical entities have digital counterparts and virtual representation. Things become context aware and they can sense, communicate, interact, exchange data, information, and knowledge [4].

From a technical point of view, IoT can also be defined as follows:

IoT is the network which can achieve interconnection of all things anywhere, anytime with complete awareness, reliable transmission, accurate control, intelligent processing, and other characteristics by the supportive technologies, such as micro-sensors, RFID, wireless sensor network technology, intelligent embedded

technologies, Internet technologies, integrated intelligent processing technology, and nanotechnology [3].

Explosive growth of smart phones and tablet PCs brought the number of devices connected to the Internet to 12.5 billion in 2010, while the world's human population increased to 6.8 billion, making the number of connected devices per person more than 1 (1.84 to be exact) for the first time in history [1]. According to [1], the number of connected devices in Internet is expected to reach 25 billion by 2015.

Due to the advances in sensor technology, sensors are becoming more powerful, cheaper, and small stimulating a large-scale deployment [5]. Advances in sensor data collection technology, such as pervasive and embedded devices, and connectivity technology such as RFID, will raise the above numbers to trillions sooner, which are connected to the Internet and continuously transmit their data overtime.

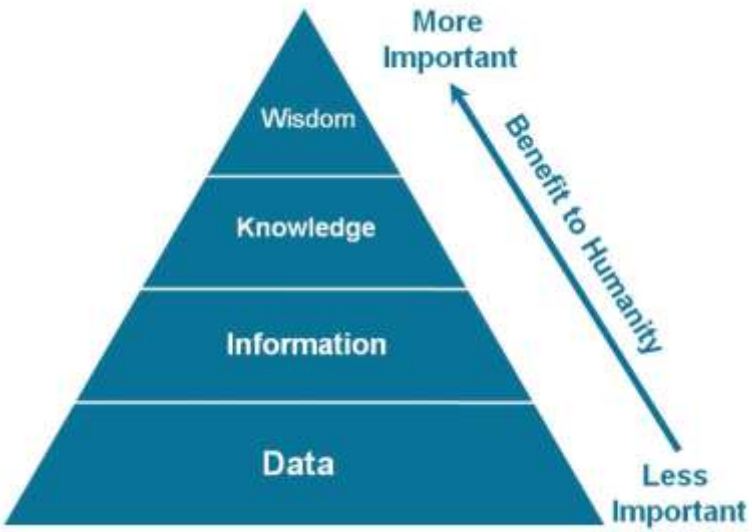

Figure 2.2: Human converts data into wisdom [1].

According to Figure 2.2, it is worthy to note a direct correlation between the input (data) and output (wisdom). The more data created, the more knowledge and wisdom people can obtain [1], which will enable people to advance even further. Ultimately, these sensors will generate big data. However, the data we collect may not have any value unless we analyze, interpret, and understand it. The traditional application-based approaches (i.e., connect sensors directly to applications individually and manually) for data collection becomes infeasible [5].

2.1.2 Challenges and Barriers to IoT

Several challenges and barriers are expected to potentially slow down IoT deployments those include:

Understandability of data: With the advances in sensor technologies, sensors are expected to attach to every object around us. These sensors can communicate with each other with a minimum human intervention. Understanding sensor data is one of the main challenges it faces. This is identified as an important IoT research need by Cluster of European Research Projects on IoT funded by European Union [5].

Large amount of data: Due to large scale generation of raw data by the attached sensors, there were several issues in the data collection, data storage, and analysis, which contribute to research towards big data analysis [6].

Addressability of devices: Since IoT builds upon the ability of uniquely identify the Internet connected devices it requires a larger address space to recognize different devices distinctively. The original Internet protocol IPv4, which permits only ~4.3 billion unique address, was insufficient after several years. Fortunately, new IPv6 protocol, which is being adopted provides an address space of 2^{128} [6]. This solves the addressability issue but certainly, this slows down the adoption to IoT.

Sensor energy: Changing batteries in billions of sensor devices deployed across the planet and even into space are not possible. Sensors need to be self-sustaining with a system to generate electricity from environmental elements such as vibrations, light, and airflow.

Privacy and Security: Privacy and security are important concerns in the system, which are concerns in IoT as well. Issues of data privacy may arise during data collections well as during data transmission and sharing [6].

Standards: Much progress has been made in the area of standards, yet more is needed [1].

Considering the benefits of the IoT, such challenges and barriers will be addressed soon. As discussed in [7] some of the further challenges include massive scaling,

architecture and dependencies, creating knowledge and data, robustness, openness, and security and privacy.

2.2 Complex Event Processing

Complex Event Processing (CEP) is an emerging technology in data processing, and it is the principal technology solution for processing continuous streams in real time [8], [9]. This technology is widely referred as "Event Stream Processing" or "Stream Processing" or "Event Processing". CEP involves rules to aggregate, filter, and match low-level events, coupled with actions to generate new higher-level events from those events [9], [10].

One of the main advantages of CEP is the usage of domain-specific declarative language to perform the event processing, which commonly referred to as the Event Processing Language (EPL) [8]. A large class of both well-established and emerging applications, which include data warehouse products, real time stock trading, monitoring, surveillance, and web analytics are some that can utilize CEPs to increase the efficiency [9]–[13]. With the popularity of IoT, the sensors were placed everywhere, which generate continuous data streams, where in many cases required to be processed in near real time. CEPs are suitable alternative to these sensor applications as CEP engines are typically capable of processing thousands of complex events with different data formats in real time [13]. Dirty and fuzzy data streams resulting from sensor devices cause probabilistic events and the research papers [14] and [15] discusses different methods to deal with such uncertain data.

2.2.1 Why Complex Event Processing

Consider the following use cases:

- A fire alarm application that continuously get the reading of the temperature sensor and smoke sensor. When the temperature increasing continuously past certain temperature and the smoke detected then the application, detected as

the fire and local fire alarm activated by the application automatically and also automated message sent to fire department as the fire detected in this address.

- A smart traffic light system in a junction that uses the average number of vehicles with in last X minutes from each road, and the current date and time as the input to calculate the time duration for the green light for each road.

The common requirement of above applications are that they need to continuously collect, process, and analyze data in real time, producing results without delay, even when the data arrives at very high rates [10].

A traditional relational databases are designed to collect and store data, which one can subsequently analyze to filter, combine, group it, search for patterns, derive high level summary data [10]. In traditional databases, the user explicitly runs the query to obtain the result. However, the event processor (the heart of CEP technology), in contrast to the traditional database, receives incoming messages and runs them through a set of pre-defined continuous queries to produce derived streams or set of data [10].

2.2.2 CEP Applications and Functions

Regardless of the specific terms used, the event processing applications typically perform one or more of the following:

1. *Situation Detection:* Monitor incoming events to detect patterns that indicate the existence of opportunity or a problem.
2. *Data aggregation and Analysis/Continuous computation:* Data is correlated, grouped and aggregated, and computations are then applied to produce new information such as summary data, and high-level statistics.
3. *Data collection:* A byproduct of CEP application often the collection of raw data and/or higher-level summary data.

4. *Application Integration, Intelligent Event Handling:* CEP can provide intelligence within an event driven architecture to analyze events and knowledge of various systems to determine what new events to generate or to determine the action to take based on an event.

While describing incoming data analyzes in real time, it actually refers to a variety of functions that can be applied to data, alone or in combination, to derive high-level intelligence and/ or trigger a response. Common functions include the following:

- Filter data to apply simple or complex filters to detect conditions of interest
- Combine data from multiple sources that arrives at different times
- Group and aggregate data, producing high-level summary data and statistics
- Transform data format and structure
- Generate high-level events from patterns or sequence of events.

2.2.3 Siddhi CEP [9]

Siddhi is a lightweight, easy-to-use open source CEP under Apache Software License v2.0. Siddhi CEP processes events that are triggered by various event sources and notifies appropriate complex events according to user specified queries [16]. Siddhi combines following design decisions to improve the performance:

- Multi-threading
- Queues and use of pipelining
- Nested queries and chaining streams
- Query optimization and common sub query elimination

As illustrated in Figure 2.3, Siddhi receives events from event sources through Input adaptors and converts them to a common data model: tuple. For example, if an XML arrives at Siddhi, its input adaptor converts this to tuple for internal processing.

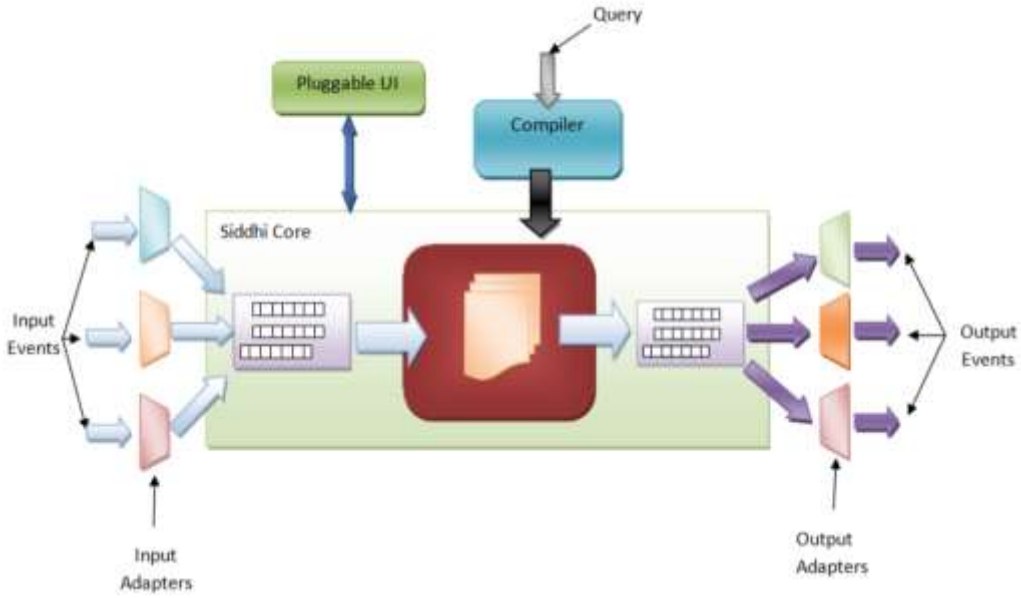

Figure 2.3: Siddhi system architecture [9].

When user submits the query to Siddhi, the Query Compiler converts the query to a run time representation (Processors) and deploys that to Siddhi core. Siddhi uses pipeline model where it breaks the execution into different stages (through Processors), and moves data through the pipeline using publication-subscription model.

Siddhi evaluates the tree in Depth First Search (DFS) order, and Siddhi optimizes this process by terminating execution whenever there are sufficient conditions to know that tree will not evaluate to true. Pattern queries and Sequence queries uses state machines to support its implementation.

Siddhi architecture allows manipulating queries on the fly, thus allowing users to add or remove queries while Siddhi engine is running, and Siddhi supports duplicate event detection.

The paper [9] evaluates Siddhi with Esper and the results indicate that Siddhi is better than Esper. Esper is the most widely used open source CEP engine and is utilized as core for many other CEP engines like Oracle.

It seems Siddhi is capable of processing 100K+ events/sec over network hardware by using 4 core 4GB CPU [16]. Siddhi is capable of processing 6M events/sec when events generate from the same JVM [16].

2.2.3.1 Siddhi Query Language [9], [16]

Siddhi query Object model follows an SQL-like query structure, which fall in line with relational algebraic expressions. Due to this reason, Siddhi queries can also utilize optimization techniques that are used in SQL and relational database. Each Siddhi query produces a stream, which can pass to another query as an input stream to create complex queries. In Siddhi, since each query's output can fed into many queries, the repetition of the same query will be eliminated.

Siddhi Queries describe how to combine existing event streams to create new event streams. For example, let's consider an event stream called '*RoomClimate'* that has 2 parameters as 'temp' of type FLOAT, and 'humidity' of type INT.

```
from RoomClimate[temp>28]
select temp, humidity
insert into HighTempValues
```

The above query will create a new stream called '*HighTempValues'* that has two attributes as 'temp' of type FLOAT and 'humidity' of type INT having events from '*RoomClimate'* that have the temp attribute greater than 28.

The Figure 2.4 shows the abstract BNF-based definition for Siddhi Query Language. Siddhi support several query types such as Filters, Windows, Joins, Patterns, and Sequences.

```
<execution-plan> ::= <define-parition> | <define-stream> |
<define-table> | <execution-query>
<define-partition> ::= define partition <partition-id> by
<partition-type> {, <partition-type>}
<define-stream> ::= define stream <stream-name> <attribute-
name> <type> {<attribute-name> <type>}
<define-table> ::= define table <table-id> ( <attribute-name>
<type> {, <attribute-name> <type>} ) { from <table-
type>.<datasource-name>:<database-name>.<table-name>}
<execution-query>::= <input> <output> [<projection>]
<input> ::= from <streams>
<output> ::= ((insert [<output-type>] into <stream-name>) |
(return [<output-type>]))
<streams> ::= <stream>[#<window>]
|   <stream>#<window> [unidirectional] <join> [unidirectional]
<stream>#<window> on <condition> within <time>
|   [every] <stream> -> <stream> ... <stream> within <time>
|  <stream>, <stream>, <stream> within <time>
<stream> ::= <stream-name> <condition-list>
<projection> ::= (<external-call> <attributelist>) |
<attributelist> [group by <attribute-name> ][having
<condition>]
<external-call> ::= call <name> ( <param-list> )
<condition-list> ::= {'['<condition>']'}
<attributelist>::=(<attribute-name> [as <reference-name>]) | (
<function>(<param-list>) as <reference-name>)
<output-type> ::= expired-events | current-events | all-events
<param-list> ::= {<expression>}
<condition> ::= ( <condition> (and|or) <condition> ) | (not
<condition>) | ( <expression>
(==|!=|>=|<=|>|<|contains|instanceof) <expression> )
<expression> ::= ( <expression> (+ | - | / | * | %)
<expression> ) | <attribute-name> | <int> | <long> | <double>
| <float> | <string> | <time>
<time> ::= [<int>( years | year )]  [<int>( months | month
)] [<int>( weeks | week )] [<int>( days | day )] [<int>( hours
| hour )] [<int>( minutes | min | minute )] [<int>( seconds |
second | sec )]
[<int>( milliseconds | millisecond )]
```

Figure 2.4: Abstract BNF representation of the Siddhi Query Language

2.2.4 Popular open source CEP engines

This section describes its architecture, feature, and advantages and disadvantages of some well-known Complex Event processing engines in the market. Some of the implementations are only a prototype implementation for a resource-limited device, some are CEP engines, while others are event-processing engines where CEP is included. Some of the state-of-art event processing systems are discussed in [17].

2.2.4.1 Esper, NEsper [18], [19]

EsperTech's complex event processing and event series analysis software turns large volume of disparate event series or streams into actionable intelligence, which is available under GPL v2 license (General Public License). It also provides other licensing options such as OEM license for ISVs (Independent Software Vendor) and commercial licensing for Enterprise editions.

Esper provides a rich Event Processing Language (EPL) to express filtering, aggregation, and joins, possibly over sliding windows of multiple event series. It also includes pattern semantics to express complex temporal causality among events. Events supports wide variety of representations such as Java beans, XML document, legacy classes, or simply name value pair.

Esper is integrated into Java and .NET languages, and can be embedded into existing middleware systems as a library. Its POJO (Plain Old Java Object) based programing model and core API enables any Java Developer to enrich an existing application with event series intelligence now.

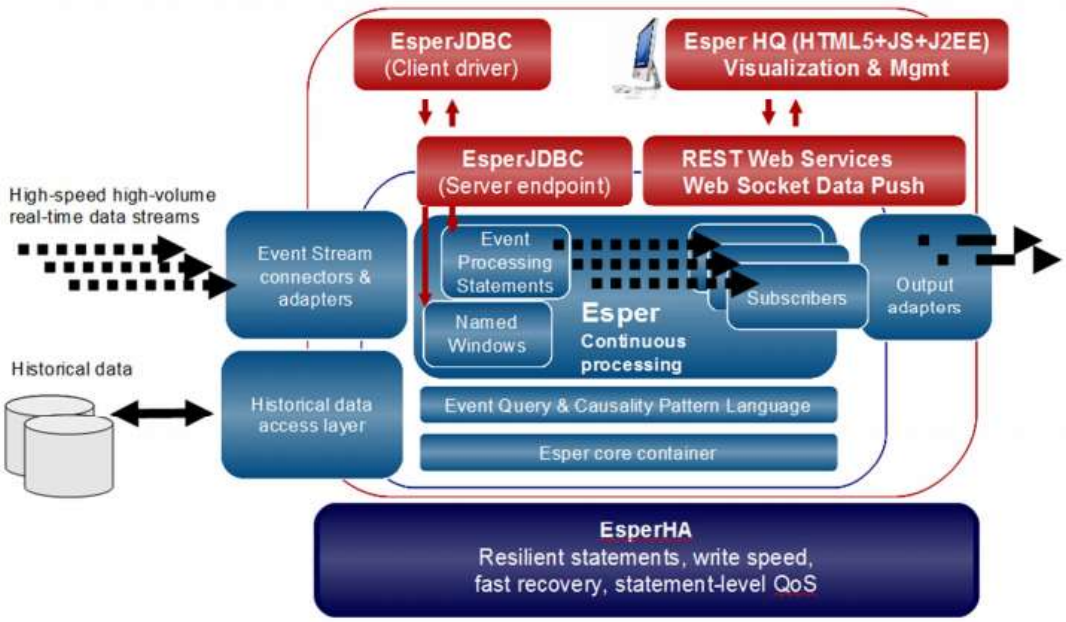

Figure 2.5: Esper Architecture diagram [18].

In Figure 2.5, the components in red color are the additional components for the Enterprise editions, besides the basic components (in blue color), which contains the rich web based user interface for real time event displays that provides CEP engine management, Design and Debug EPL statements, and hot deployment.

2.2.4.2 Aurora [20][21]

Aurora addresses three broad application types in a single, unique framework. For instance, Real-time monitoring applications continuously monitor the present state of the world and interested in the most current data as it arrives from the environment, while archival applications typically interests in the past, and spanning applications involve both present and past states of the world required to combining and comparing incoming live data and stored historical data.

Aurora processes tuples from incoming streams according to a specification made by an application administrator. Aurora is fundamentally a data-flow system and uses the popular boxes and arrows paradigm found in most process flow and workflow systems.

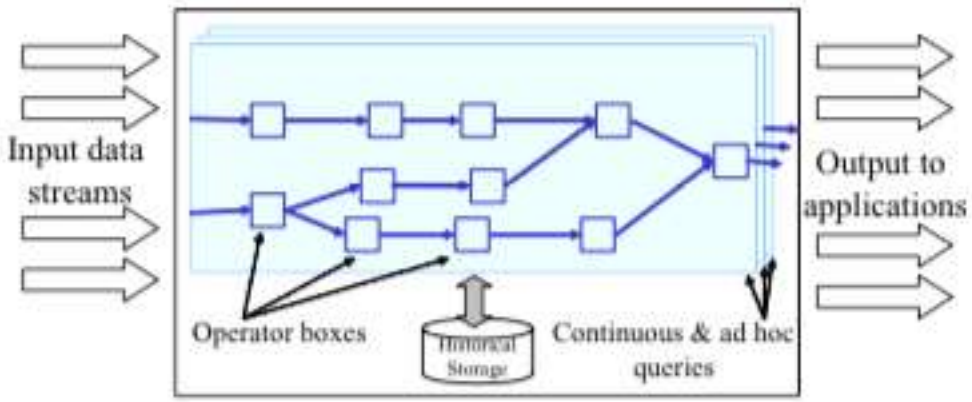

Figure 2.6: Aurora system model [20].

Here tuples flow through a loop-free, directed graph of processing operators as shown in Figure 2.6. Every Aurora application must be associated with a query that defines its processing requirements, and a Quality of Service (QoS) specification that specifies its performance requirements.

Queries are built from a standard set of well-defined operators (boxes). Each operator accepts input streams (in arrows), transforms them in some way, and produces one or more output streams (out arrows). Heart of the system is scheduler that determines the number of tuples that might be waiting in front of the given box to process and the distance to push them toward the output.

This architecture splits the general problem into intra-participant distribution (relatively small-scale distribution all within one administrative domain, handled by Aurora) and inter-participant distribution (large-scale distribution across administrative boundaries, handled by Medusa).

Medusa is a distributed infrastructure that provides service delivery among autonomous participants. To handle this architecture efficiently, the system should achieve three goals such as a scalable communication infrastructure, adaptive load management, and high availability.

2.2.4.3 Cayuga[22], [11]

Purpose of designing and building Cayuga as a general-purpose system, was to process complex events on a larger scale. It supports online detection of many complex patterns in event streams. Cayuga event language based on Cayuga algebra was designed for expressing queries over event streams. It is a simple mapping of the algebra operators into a SQL like syntax.

Figure 2.7 describes the Cayuga System Architecture. Event Receivers (ERs), each of which runs in a separate thread, receive external events. ER threads are responsible for de-serializing arriving events, assigning time stamps if necessary, internalizing them in the Cayuga Heap and inserting them on the input Priority Queue (PQ). Cayuga Query engine is a single thread responsible for a majority of query processing work. The engine de-queues events from the PQ in detection time order and performs all indicated automation state transition. For each automation instance reaching final state, it enqueues a new event on the PQ if required for re-subscription, and passes events to the appropriate Client Notifier threads (CNs).

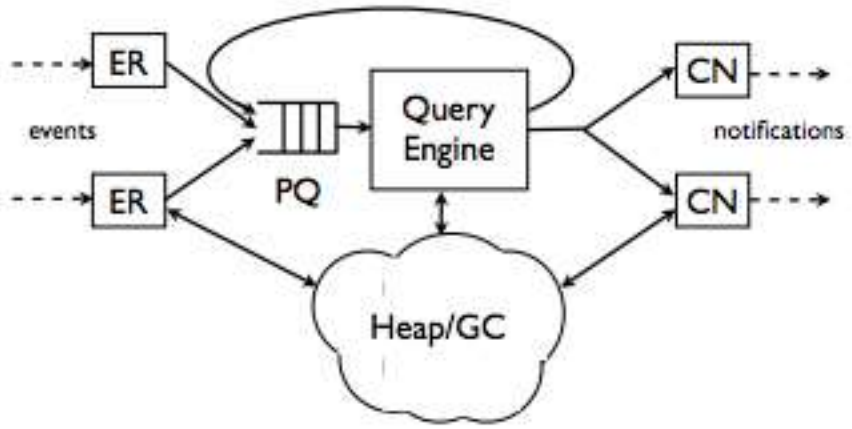

Figure 2.7: Cayuga system architecture [11].

Cayuga uses two methods for memory management such as garbage collection by periodically destroying unnecessary data, and uses Internal String table to manage read-only string objects stored in the Cayuga Heap to ensure that there is at most one copy of any string value in the heap.

Cayuga can output a continuous trace of how its internal state changes between events. This trace is written to a file, and Trace Visualizer reads the trace file and uses Java Swing based GUI to display how events are matched.

2.2.4.4 PIPES [23]

PIPES is a flexible and extensible infrastructure providing fundamental building blocks to implement a Data Stream Management System (DSMS).

The core framework allows constructing directed acyclic query graphs based on publish-subscribe mechanism integrated into the graph nodes. As illustrated in Figure 2.8, source transfers its elements to a set of subscribed sinks. A sink can subscribe and unsubscribe to multiple sources respectively. During its subscription, it processes all incoming elements delivered by its sources. All operators satisfy the interface pipe that combines the functionality of a sink and a source. Hence, a pipe processes the incoming elements and transfers its outgoing elements to all subscribed sinks.

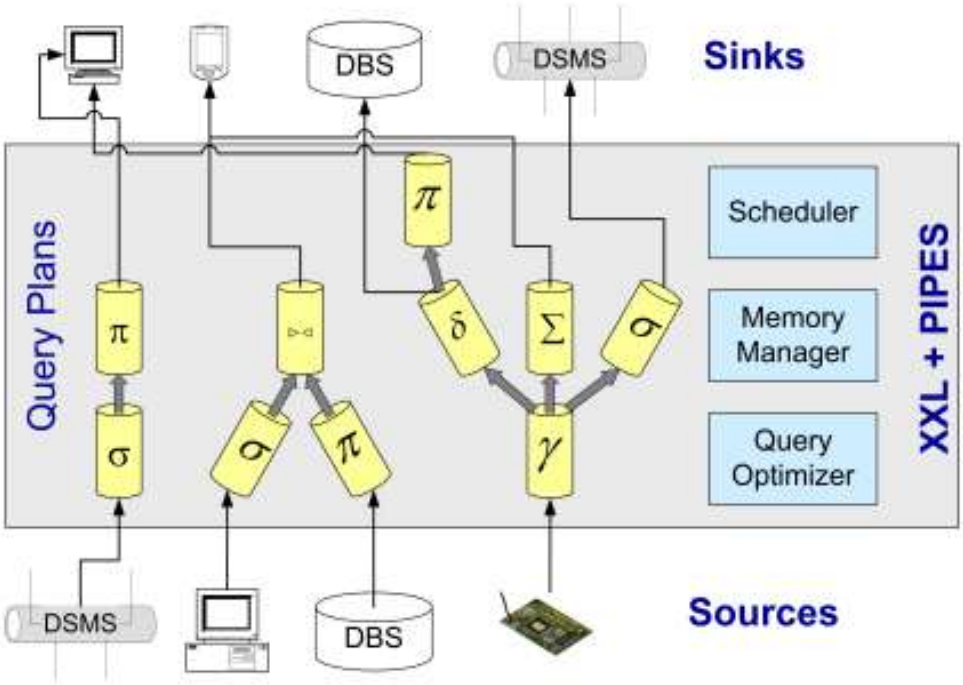

Figure 2.8: PIPES Architectural overview [23].

2.2.4.5 SASE [13]

SASE is a complex event processing system, designed and developed to transform real-time RFID data into meaningful, actionable information. SASE language has a high-level structure similar to SQL for ease of use, but the language design is centered on event pattern matching.

The architecture of SASE system consists of three layers as presented in Figure 2.9. The bottom layer contains physical RFID devices called 'Physical Device Layer'. The RFID returned from RFID readers is passed to the next layer called 'Cleaning and Association Layer' for data cleaning and event generation. This layer first performs data cleaning, such as filtering and smoothing. This is important, as RFID readings are known to be inaccurate and messy. Second, it uses attributes such as product name, expiration date, and saleable state to create events. This helps facilitate processing and decision making in subsequent components.

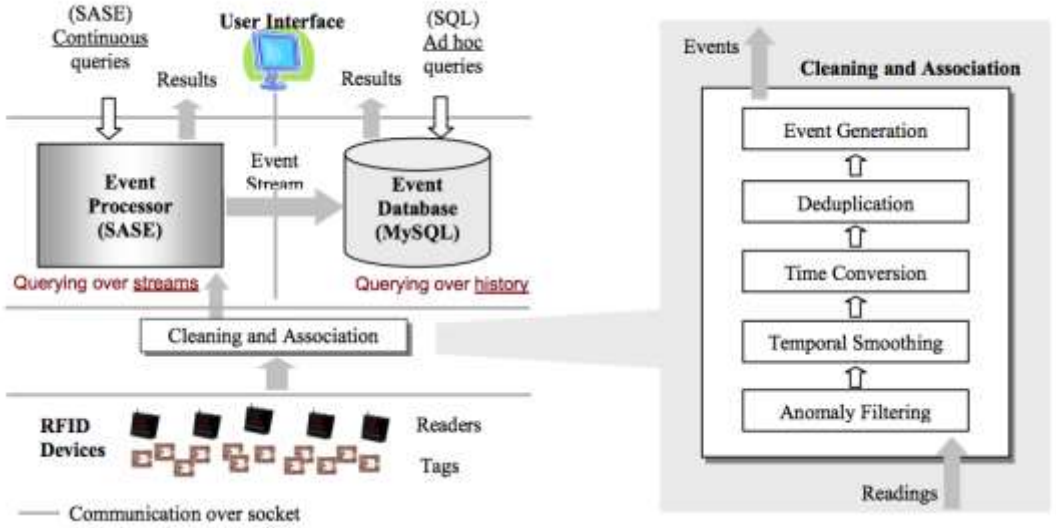

Figure 2.9: SASE system architecture [13].

2.2.5 Related work on Light-weight CEP engines

No lightweight CEP system is available that runs on embedded devices. However, several related work towards implementing the light-weight CEP engine or work related to light-weight CEP engine are discussed in this section.

2.2.5.1 Concurrent Reactive Objects (CRO) model [2]

This section describes the architecture of the designing method of this model as a lightweight complex event processing using the concurrent reactive object (CRO) model. The core feature of this model is to react to atomic events. Between the reactions/function executions, the system remains idle, and thus abstains from occupying the CPU and is energy-efficient.

An event query language called CEDR language, where CEDR is a declarative language to express queries over event streams, is used to express the event patterns. General form of a CEDR query includes:

```
EVENT <query name>
WHEN <event expression>
WHERE <correlation expression>
OUTPUT <instance transformation conditions>
```

The CRO model is the execution and concurrency model of the Timber programming language, which is a general-purpose object oriented language that primarily targets real time systems. A subset of C implementing the core features of Timber and using the CRO model as its execution model is called Tiny Timber. CRO model facilitates reactivity, object-orientation with complete state encapsulation, object level concurrency with message passing between objects, the ability to specify timing behavior of the system, and the abstraction to the components.

Reactivity is the defining property of the CRO model, which makes it particularly suitable for embedded systems, since functionality of almost all embedded systems can be expressed in terms of reactions to external thing/event and timer events.

Implementation of an object instance can be either software or provided by the environment. This allows incorporating hardware interactions and legacy code in the model as long as their interface is complaint with the current reactive object model.

The idea is based on compile-time translation of a CEP query expressed in the CEDR query language, into a set of concurrent reactive object. Interconnections between the objects reflect logical links between sub-expressions of the query. This is an ongoing work and future work is outlined in [2].

2.2.5.2 CEP Technology stack on Gumstix [24]

This paper describes the working stack of semantic technologies for reasoning enabled CEP as proof of concept (POC) implementation on the Gumstix embedded controller. Gumstix embedded controller is a very small general-purpose computer made for ubiquitous computing applications that runs a Linux operating system.

Based on Linux-driven Gumstix platform the paper established a stack of technologies to realize embedded situation recognition as shown in Figure 2.10. Processor Architecture is the bottom-most layer of target processor architecture. Prolog Layer is on top of the Linux system that runs on Gumstix, Prolog engine is used as a basis to perform rule based reasoning. CEP is layered on top of Prolog.

Situation Recognition Layer's purpose is to merge all the technologies involved for realizing knowledge based recognition applications.

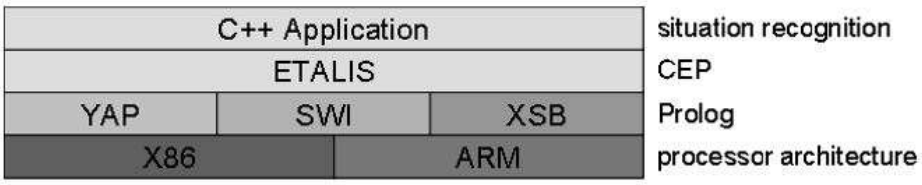

Figure 2.10: Software Technology stack [24].

2.2.5.3 LiSEP [25]

LiSEP (Lightweight Stage-based Event Processor) design has been driven by ease-of-use, extensibility, scalability, and portability requirements. LiSEP is based on a layered architecture, whose design clearly separates the core logic devoted to event processing from low-level thread management handled by the Staged Event-Driven Architecture (SEDA) framework. The design has been driven by the principle of minimizing dependency on external software components. In addition, LiSEP depends solely on the Java Standard Edition libraries, thus minimizing deployment requirement. Moreover, the LiSEP logic strictly focused on core event processing, thus resulting in a lightweight and minimal implementation leaves the overall JAR package is limited to 360Kbytes. The LiSEP Event Processing Language provides an expressive and user-friendly querying modelling capability, based on an SQL-like syntax.

In a typical deployment configuration, input events are generated by external applications and routed to CEP engine by proper messaging infrastructure such as Enterprise Service Bus. Input data can be delivered by sources in different message formats such as XML, JSON and then transformed into the java-based internal representation by proper adaptor components. Query statements are used to express target event patterns. When specific event pattern is detected, registered listeners are notified and execute specific actions as a reaction to successful even detection as shown in Figure 2.11.

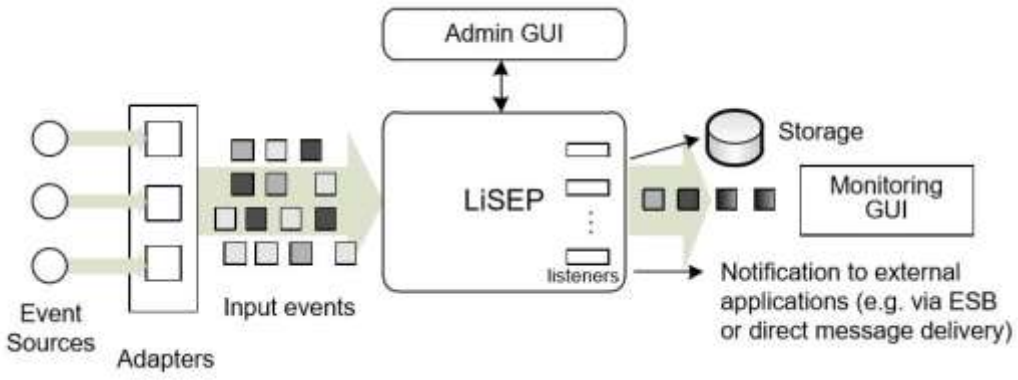

Figure 3.11: LiSEP event processing flow [25].

LiSEP is built around a set of features aiming to achieve portability, modularity and extensibility, scalability, and minimal configuration and deployment requisites.

As shown in Figure 2.12 the LiSEP tool is composed of two layers. The upper LiSEP event-processing layer deals with event evaluation and processing tasks and is implemented as a graph of event-driven stages connected with explicit event queues in accordance to the SEDA architecture. It encloses the core logic of LiSEP engine and leverages on stage building capabilities provided by the lower SEDA framework. The other layers represent an abstraction of the execution environment.

LiSEP stages are such as Statement builder, Statement manager, Clause manager, Listeners manager, Message-based stage interaction, etc.

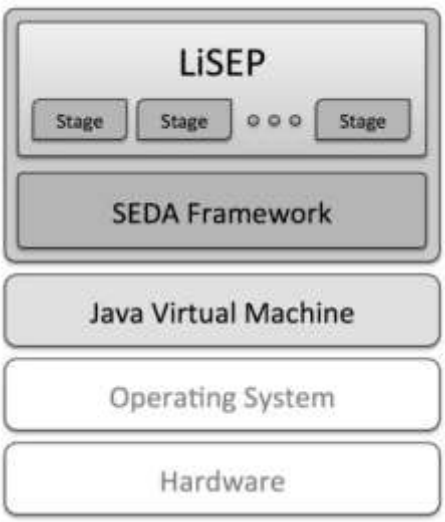

Figure 2.12: LiSEP layered architecture [25].

2.2.5.4 Esper [26]

Esper is a lightweight kernel written in Java, where Esper engine is a JAR, and instantiated by the means of a Java API and an XML configuration file.

This is the common ground for 3 editions (Esper, EsperEE, and EsperHA) and where the core event processing logic is implemented. It comes in two flavors, which are Esper for Java and NEsper for .NET, which are embeddable components written in Java and C# and are therefore suitable for integration into any Java process of .NET-based process including J2EE application servers or standalone Java applications. Esper and NEsper are not a server by itself but are designed to hook into any sort of server, ranging from market standard J2EE server (weblogic, websphere, jboss, etc.), service bus, or lightweight solutions (OSGi based and grid) and also Microsoft based .Net technologies. NEsper is suitable for use in desktop end-user stations.

2.2.5.5 Triceps [27]

Triceps is open source and implemented in C++ and Perl scripting language. Therefore, nothing really prevents embedding Triceps into other languages. No separate server executable, no need to control it, and no custom network protocols: the users can put the code directly into their executables and devise any protocols they please.

This is the CEP engine as an in-memory database driven by triggers, a data-flow machine, or a spreadsheet on steroids (and without the GUI part). All the C++ code has been written with multithreading in mind, however for the first release the multithreading did not propagate into Perl API yet.

2.2.5.6 Complex Event Detection with FPGAs [28]

The challenge for many complex event processing (CEP) systems is to be able to evaluate event patterns on high-volume data streams while adhering to real-time constraints. This paper presents a hardware based, complex event detection system

implemented on field-programmable gate arrays (FPGAs). By inserting the FPGA directly into the data path between the network interface and CPU, enables to detect complex events at gigabit wire speed with constant and fully predictable latency, independently of network load, packet size, or data distribution. This solution uses regular expressions implemented as finite automata to detect complex events.

2.3 Embedded Devices

2.3.1 Why CEP with Resource Limited hardware

Industrial process automation systems are adopting event-based communication that pushing control loops towards low-level devices requires for lightweight embedded devices that are able to recognize and react to events [2].

Atomic events such as value read by an individual sensor exceeding certain value is inadequate. Rather, it requires capturing scenarios where a reaction should occur in a sequence of low-level events matching certain pattern [2]. Therefore, it is desirable that resource-constrained low-level devices are equipped with some possibly lightweight form of event filtering and processing [2], [24]. This lightweight form of event filtering and processing technology can be called as Simple CEP.

These applications often need to run on embedded devices that fit into an industrial environment and can, e.g., be placed close to the sources of sensor signals [24]. Since different industrial environment physical conditions vary, i.e., some embedded devices placed near very heat while some operates outside in rain, wind, and snow, this requirement opens up the need of open source hardware, as industry can design or manufacture suitable embedded devices for their specific needs.

2.3.2 Open Source Hardware

Rapid advances in electronic technologies have resulted in a variety of new and inexpensive sensing, monitoring, and control capabilities. These rapidly evolving

technologies provide researchers and practitioners with low-cost, solid-state sensors and programmable microcontroller-based circuits [29].

Free and open source hardware (FOSH or Open hardware) is shared by providing the bill of materials, schematics, assembly instructions, and procedures needed to fabricate a digital replica of the original. FOSH benefits from mass-scale peer reviews and collaborations, which has been proven to be successful in free and open source software [30].

The literature survey searched for microcontroller based open-hardware due to above advantages, which will encourage researchers and practitioners to use in various applications and make this a successful application.

Creation of a micro controller based development platform called Arduino is the first large-scale success in open source hardware [31].

2.3.2.1 Arduino [32]

Arduino is a tool for making computers that can sense and control more of the physical world than the desktop computer [32]. Arduino hardware consists of programmable microcontroller mounted on input/output pins and connectivity to personal computer for programming and user interaction. Since Arduino circuit board has a standardized physical configuration, any Arduino compatible board can use interchanged. There are several standardized add-on boards called ‘shields’, that can enhance the Arduino main board capabilities [29].

In brief, Arduino Uno is a most popular microcontroller board based on ATmega328, containing 32 kilobytes (KB) of flash memory for program storage, and 2 KB of non-volatile memory [32]. The microcontroller contains many built-in features including timer/counter, internal/external interrupts, and serial and other communication protocols. The software environment for programing and interacting with Arduino board is available for download that has installers for main operating systems. Using

this IDE, users can write programs in a language based on C++, compile and error check the program, and download compiled routine to the microcontroller [29]. There were several Arduino boards currently available, Table 2.1 lists ten popular boards among them and compares its configurations in a tabular format.

Figure 2.13, explains main components of the Arduino UNO board. A USB cable or a power supply in a barrel jack can power Arduino boards. USB connection can also load the code to Arduino board and the recommended voltage of many Arduino boards between 6V and 20V. The Power LED should light up whenever Arduino plug into a power source. Arduino has a Reset button that will temporarily connect the reset pin to ground when pushed, and restart any code loaded on Arduino.

Table 2.1: Configuration comparison for ten popular Arduino boards [32].

Board Name	**Processor**	**CPU speed**	**Analog Input**	**Digital IO/PWM**	**SRAM (KB)**	**Flash (KB)**
UNO	ATmega328	16MHz	6	14/6	2	32
Due	AT91SAM3X8E	84MHz	12	54/12	96	512
Leonardo	ATmega32u4	16MHz	12	20/7	2.5	32
Mega 2560	ATmega2560	16MHz	16	54/15	8	256
Mega ADK	ATmega2560	16MHz	16	54/15	8	256
Micro	ATmega32u4	16MHz	12	20/7	2.5	32
Mini	ATmega328	16MHz	8	14/6	2	32
Nano	ATmega168	16MHz	8	14/6	1	16
	ATmega328				2	32
Ethernet	ATmega328	16MHz	8	14/4	2	32
LillyPad	ATmega168V	8MHz	6	14/6	1	16
	ATmega328V					

The pins in the Arduino are the points where we connect wires to construct the circuits. GND is short for ground and can be used to ground our circuit. The 5V pin supplies 5 volts of power and the 3.3V pin supplies 3.3 volts of power. AREF stands for Analog reference, which is occasionally used to set an external reference voltage (0 - 5 volts) as the upper limit for the analog input pins.

A0 through A5 are Analog In pins, which can read signals from Analog sensor and converts it into a digital value. Digital pins (0 through 13) can be used for both digital input and digital output. In addition, the tilde (~) mark next to some of the digital pins can be used for Pulse-Width-Modulation (PWM). TX is short for transmit and RX is short for receive. These pins are responsible for serial communication. Voltage regulator cannot be interacted by the user, which is to turn away an extra voltage that might harm the circuit.

Main Integrated Circuit (Main IC) is the brain of Arduino boards and slightly differs according to the board type. Usually this board arrives from the ATmega line of ICs from the ATMEL. Arduino makes several different boards, each with different capabilities such as Arduino UNO, LilyPad Arduino, Arduino Mega, and Arduino Leonardo.

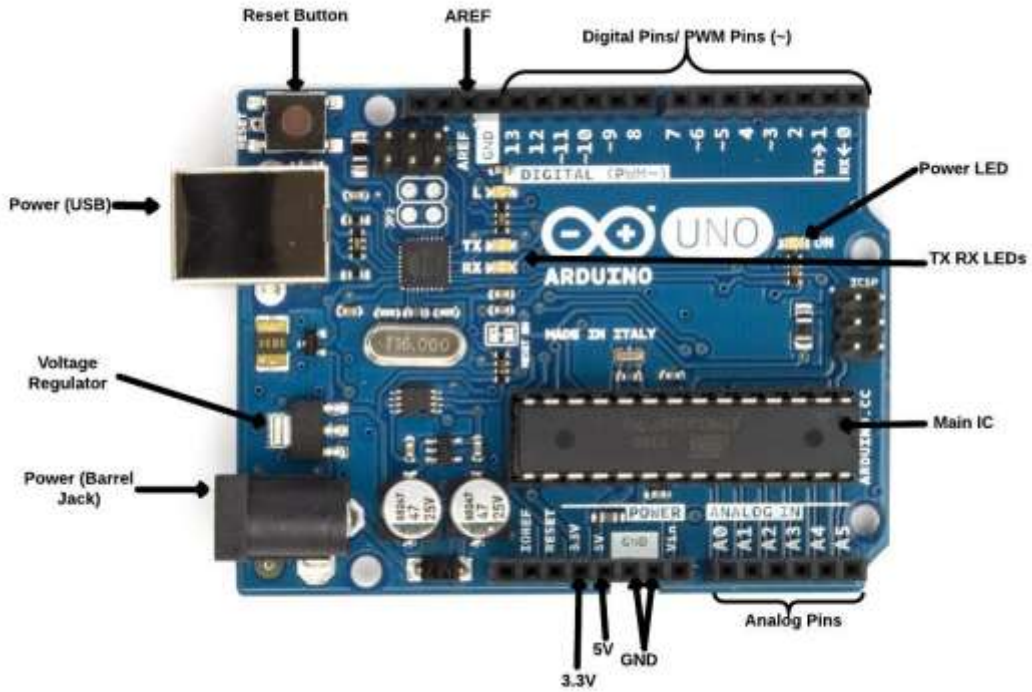

Figure 2.13: Main components of the Arduino UNO board.

Arduno board can perform many tasks independently. Basic sensors and Arduino shields can bring the projects to life. With some simple codes, the Arduino can control and interact with a variety of sensors to measure things such as light,

temperature, pressure, proximity, and acceleration. Shields are pre-built circuit boards that fit on top of the Arduino and provide additional capabilities such as controlling motors, connecting to the internet, and controlling an LCD screen.

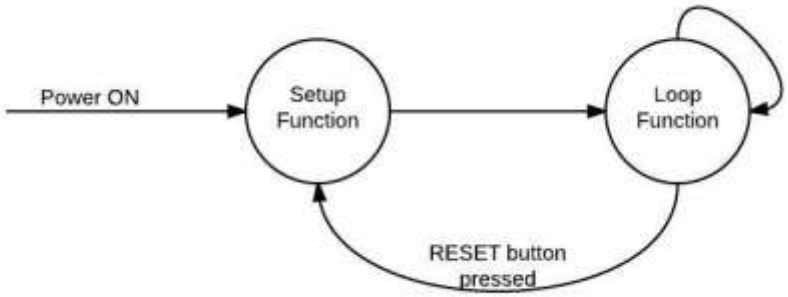

Figure 2.14: State diagram of the Arduino program [33].

Every Arduino program is made-up of a minimum of two functions. First is the Setup function that runs initially and once only, which informs Arduino, what is connected and where, as well as initializing any variables that might need in the program. Second is the loop function, which is the core of every Arduino program. When Arduino is running, after completing the setup function, the loop will run through all the codes, and then repeat the entire process repeatedly until either the power is lost or the reset switch is pressed. Figure 2.14 illustrate the state diagram, which demonstrates the Arduino Program.

As an open source project, Arduino benefits from the collective efforts and expertise of developers globally. Programming libraries, which contain routines to simplify programing and incorporated advanced features, sample codes, and complete programs, are accessible through Arduino project website, that can download, use, and modify as needed [29].

2.3.2.2 TI Launchpad MSP430 [34]

TI Launchpad MSP430 is designed at a minimum cost and low power consumption embedded applications. Launchpad also supports USB cable for programing and

power. This is amazingly energy efficient, and can use with battery for a longer time. It has a power saving mode, in which it uses virtually no power but can wake up and returned to full power in one microsecond.

It has 16 IO pins where eight can do analog and eight can do digital, where 7 out of 8 digital supports PWM. Likewise, it has processor power comparable to Arduino. Similar to Arduino shields, Launchpad also has expansion boards called 'Booster Packs'.

In comparing to Arduino UNO basic device, Launch pad's storage is 16KB, which is half the size of Arduino UNO storage. Launchpad uses 512B, which is very small compared to Arduino UNO (2 KB RAM). Finally, a massive community supports for Arduino compared to Launchpad. These above three factors make Arduino a clear favorite than Launchpad.

2.3.2.3 Wiring [35]

Wiring S board (popular board in Wiring) has more pins (32 IO pins) and more memory (64KB) compared to the Arduino UNO. Wiring was an older attempt of what Arduino successfully accomplished. It was designed to be an educational platform for learning about microprocessors, software, and physical computing. The program IDE is similar to Arduino.

Wiring board is expensive than Arduino UNO. The price, and the Arduino's vast community support, favors Arduino over Wiring for this project.

2.3.2.4 Pinguino PIC32 [36]

Pinguino is a solid prototyping tool, originally designed for art students. Despite similarities with Arduino, it is much less developed. Pinguino add on functionality compared to Arduino, Pinguino IDE is a complete rework and is not based on Wiring and this cause difficulty in switching from Arduino to Pinguino. The above discussed

facts and Arduino's massive community support makes Arduino superior to Pinguino.

2.3.2.5. Teensy++ [37]

Comparably Teensy seems better than Arduino because Teensy board is much smaller (roughly, size of a quarter), and cheaper than Arduino. Teensy supports C as a programing language, which makes it a step nearer to hardware than Arduino language. Furthermore, Teensy supports Arduino libraries and sketches.
Despite the above advantages of Teensy over Arduino, it requires experienced people in a rapid prototyping environment to get started easily compared to the Arduino, and Arduino's huge community support favors Arduino over Teensy.

2.4 Compiler Generator

The application requires a compiler/parser that generates query object model from the query. Thus, a compiler generator was explored as part of the literature survey. One of the popular and easy-to-use compiler generator is ANTLR.

2.4.1 ANTLR

ANTLR (Another Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files [38].

ANTLR is a free and an open source. In addition, ANTLR generated parsers automatically builds convenient representation of the input called *parse trees* that an application can walk to trigger code snippets as it encounters constructs of interest. Furthermore, the ANTLR has plugins for popular IDE (Integrated Development Environment). Many worldwide users apply ANTLR for their needs. Hence, there are high chances to identify and correct the bugs as early as possible.

Chapter 3

DESIGN

In chapter 1, we demonstrated the ideal state for the IoT. First, we demonstrate a model that reliably addresses this ideal scenario in this chapter.

As the proposed Complex Event Processing (CEP) engine is to be developed for embedded devices, the key design constraints include a smaller footprint (aka. code size), minimum dynamic memory consumption, and low processor overhead. It is imperative to adhere to these constraints as embedded devices typically have a very little memory (both Flash memory and RAM) and less powerful microcontrollers/CPUs. Moreover, the proposed CEP engine must be designed and developed for the Arduino architecture for it to be useful across a range of open source hardware systems.

This chapter provides a detailed explanation of the design of the CEP engine for embedded devices (CEED) and implementation details are discussed in the next chapter. Section 3.1 discusses the system architecture, whereas Section 3.2 presents the use case, architectural, and process diagrams. Section 3.3 discusses the major design consideration.

3.1 Proposed Architecture

We propose the solution illustrated in Figure 3.1 to achieve the ideal scenario for IoT that was described in Section 1.1. As shown in the figure the user will only require supplying the query the proposed CEP engine should run, and defines the Input/Output adaptor related implementation details in web interface. The web app will generate a ZIP folder that consists of Compiler, Settings.xml, automated script, and CEP libraries. The proposed solution will take care of all the remaining tasks of creating the CEED.

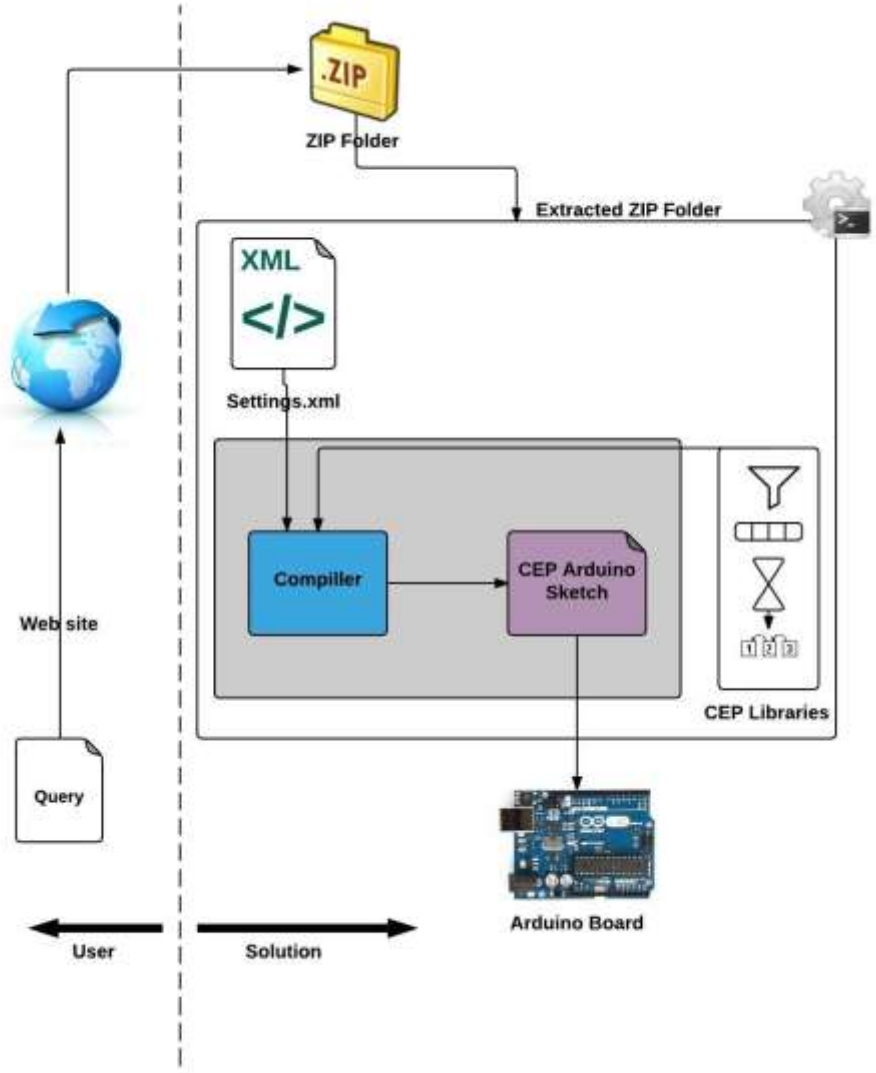

Figure 3.1: Proposed solution for CEED.

The solution consists of compiler, which takes Settings.xml, and set of CEP libraries from the package downloaded from the web site as input to generate the Arduino sketch. This Arduino sketch require to be uploaded to the desired Arduino will produce the CEED. Because all of the above tasks are automated by the script downloaded from the website, user only needs to extract the zip folder downloaded from the web site and run the script. User will now connect all the required sensors, actuators, and other event sources/receivers to create the required circuit.

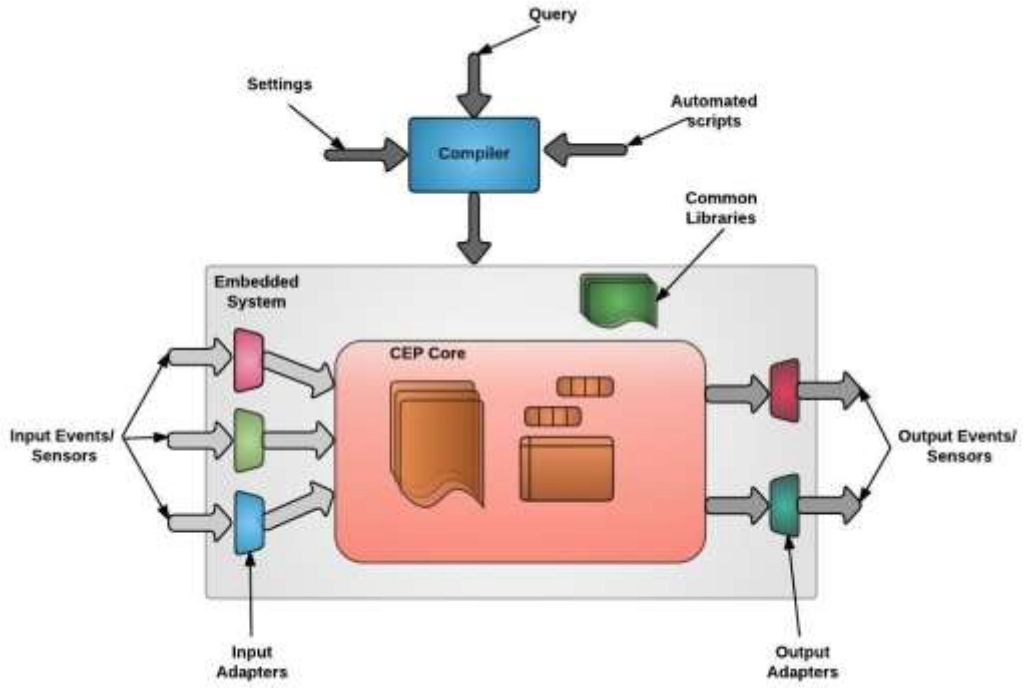

Figure 3.2: High-level architecture of the proposed CEED.

Figure 3.2 illustrates the high-level architecture of the proposed CEED. Following is a description of major components of the architecture.

Input Adapters

Major task of the input adapter is to provide an interface for event source(s) to send events to the CEP engine. Input adapters convert the events to a particular data structure used by the CEP engine for internal processing. We decided to use similar CEP Tuple data structure used by Siddhi because retrieving data will become much simpler and efficient compared to other alternatives (like XML). Users may modify the input adapter to support any custom event type.

CEP Core

Core of the CEP engine is the brain of the system where all processing takes place. Internal structure of the CEP engine core depends on the query issued to the CEP engine. For example, if the CEP engine core is built on Filter query type, then events that meet the filter condition will be transformed into output stream based tuple or unsatisfied events will be just discarded. In addition if this is built on Pattern query,

then each event will be first identifies the respective event type and checks against with the current state machine event. If the event matches the current state machine event, will be inserted. If the event is the final state machine event that completes the pattern, then new CEP Tuple created based on the output stream, and the state machine and memory will be reset. Section 3.3 provides a detailed discussion on internals of the CEP engine.

Compiler

Major task of the compiler is to create an Arduino sketch (sketch is the program file for the Arduino). The compiler takes in a CEP query and other related settings from the settings.xml file, and then generates the parse tree from the query. This parse tree is then converted to an Arduino sketch. Finally, the automated script needs to upload this sketch to the Arduino board. While the compiler is presented as a single component in Figure 3.2, it actually consists of several components.

Output Adapters

Output adapter is the inverted Input adapter, which receives the CEP tuple from CEP engine Core and then converts it back to the required format of the output actuators or streams. CEP engine returns the events in plain text format as a default behavior. However, similar to Input adapters, user can modify the output adapters to convert the tuple back to any format the event receiver needs.

3.2 Design Views

This section illustrates the system in more detail using several diagrams such as use-case, sequence, architectural, and process diagrams.

In Figure 3.3, Generate Sketch use case relates to Create Settings.xml use case. Create Settings.xml use case relates to three use cases, namely Submit Query, Assign Input Adapter, and Assign Output Adapter. Submit Query use case creates the CEP Query and include it in the Settings.xml. Assign Input adapter and Assign Output adapter are the other two use cases that incorporates correct settings to the

Settings.xml, if the input/output event type is supported by the CEP Engine. Once this Settings.xml is ready, the Generate Sketch use case generates the sketch (Arduino program) based on Settings.xml.

Submit Query and Connect Input/ Output sensors or event source are the use cases related to the Client actor. Client actor will submit the query and related details via web page. Once the sketch is uploaded to the Arduino board, Client will connect the relevant Input/ Output sensors or event source to complete the circuit.

Assign Output Adaptor and Assign Input Adaptor use cases are the optional use cases which only applies if the Client is required to do some additional advanced modifications to the Generated sketch. Direct involvement of the client in the above use cases was not shown explicitly in the use case diagram for simplicity and very rare scenario. In this event, the Client needs to interrupt the automated process to open the sketch in Arduino IDE to modify the sketch. After completing the modification, Client needs to upload the sketch to the Arduino board manually through Arduino IDE.

Actor Automated Tasks/ Compiler includes the tasks of generating the ZIP folder with automated script, Settings.xml, desktop program, and relevant library files. Generate sketch and upload sketch are the main use cases involved by Automated Tasks/ Compiler actor in addition to the generating ZIP folder. Generating ZIP folder is not shown as a separate use case in the diagram because it is still the support tasks even though this is one of the core activities.

Responsibility of the Send events use case is to receive events from the event source and send them to Input adapter to convert the event to CEP tuple format for internal processing. Event subscribers get notified of events once the output adapter converts CEP tuple to the required format and sends the event to a suitable output event receiver.

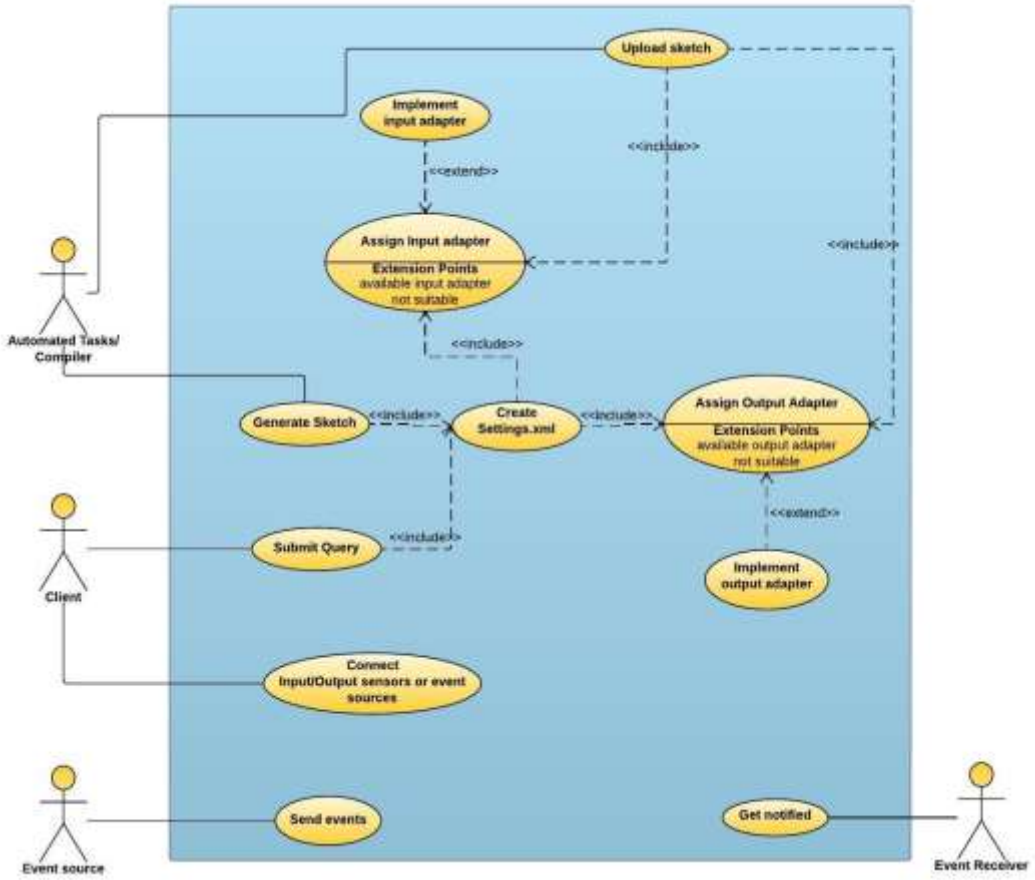

Figure 3.3: Use Case diagram of the proposed CEED.

Figure 3.4 illustrates the user flow. Compiler performs all the tasks listed under Desktop program. The first step in user flow is to open the web site, and then add the CEP query on which the CEP engine is expected to run. User can also add input and output adapters related settings in the web page. The web site will generate the relevant Settings.xml, automated script, desktop program, and add the relevant libraries to create the ZIP package.

User will download the above ZIP package and unzip in his local machine, then run the automated script found in the unzip folder. Automated script will do the series of activities like validate the Settings.xml and query, Create Sketch, and upload the sketch to the embedded device.

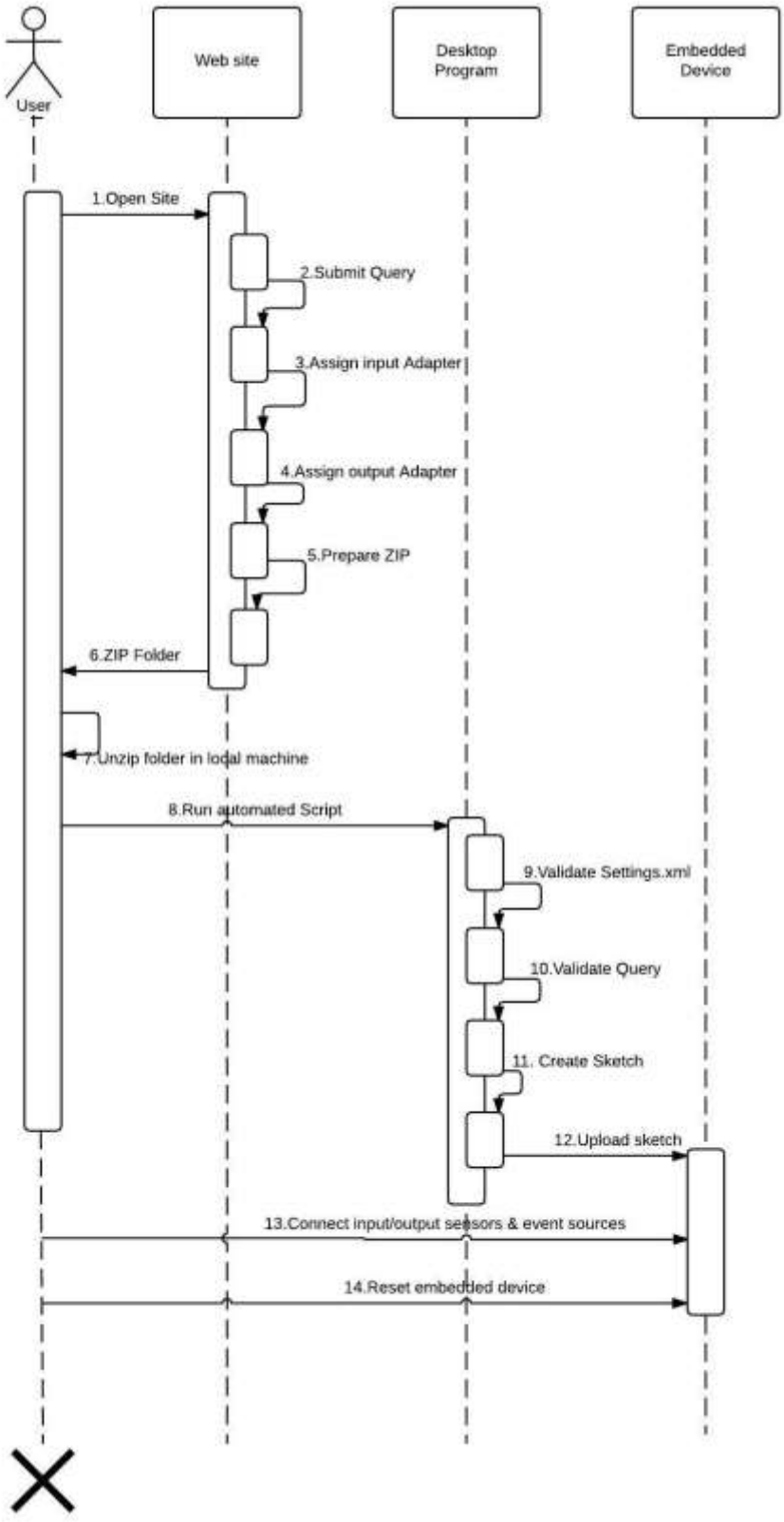

Figure 3.4: Sequence diagram for the usage sequence for proposed CEED.

Finally, the user needs to connect the input/output sensors and/or event sources and reset the Arduino board. According to the logic, when the sketch is uploaded to the Arduino board, this is ready to receive the event and process.

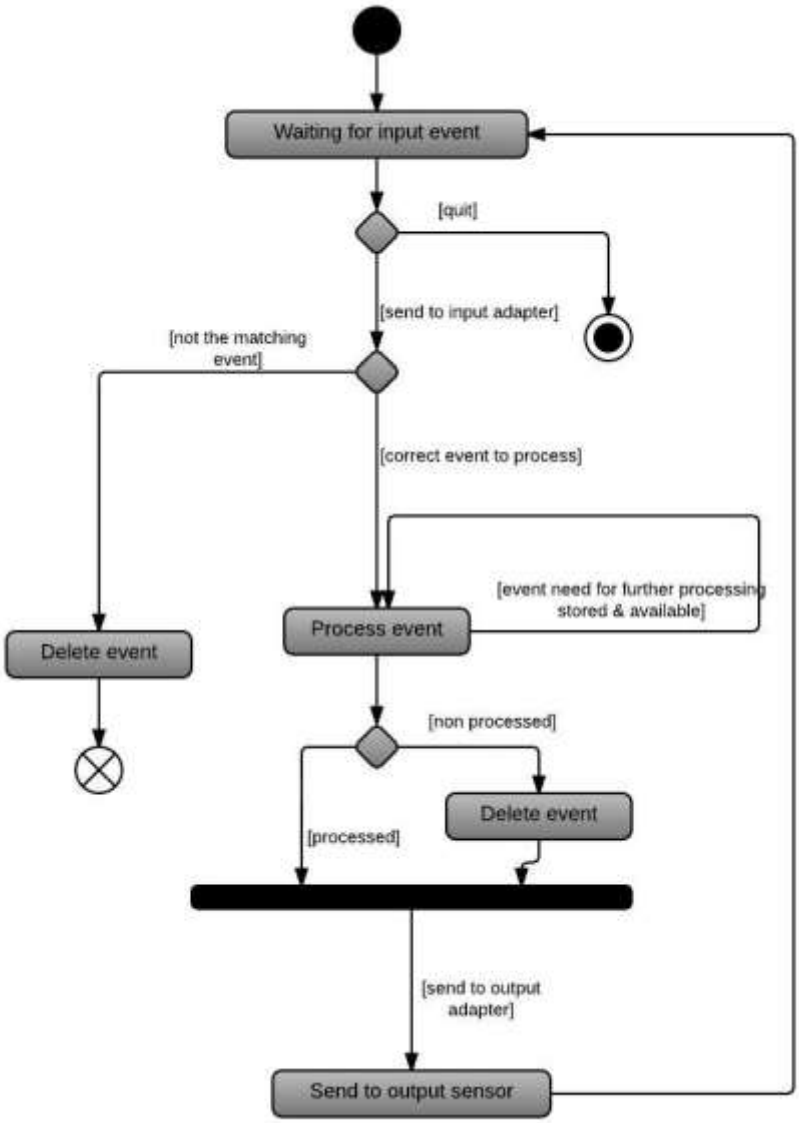

Figure 3.5: Process view of proposed CEED.

As presented in Figure 3.5, the board is actively listening to the incoming events (serial input) or idling until the next reading (digital and analog), when it is not performing any processing. As soon as an event is received to the input adapter, it converts the event to CEP Tuple, and perform processing. Finally, if the incoming event triggers and output event, the output event then converts the CEP Tuple to

event receiver's required format. Hence, each incoming event is completely processed before the CEP engine starts to process another incoming event. Parallel processing of events is not possible as Arduino only supports single-threaded programing.

Figure 3.6 also illustrates the processes discussed above, which happens in the Arduino board. Input Adapter validates the received event for correctness before initiating further processing. In case of receiving an invalid event, system will discard the said event and move back to listen for new events, which shown in both Figure 3.5 and Figure 3.6. Correct events are converted to CEP Tuple format and sent for further processing. The processing task will decide whether the event is needed to store for further processing, and if so, it will store the event in CEP's internal memory. In addition, the processing task might either discard the event if it fails to satisfy the requirement of the query, or it transform the event and notify the Output adapter if the event processed successfully.

Output adapter receives this modified tuple and converts the tuple to event receiver's required format and sent to the actual event receiver. As presented in Figure 3.5, after completing the tasks on the event, the system always moves back to listen for new events.

The deployment view (also referred to as the physical view) illustrated in Figure 3.7 provides the engineer's view of the system. Small three-dimensional boxes refer to the physical devices while other two-dimensional rectangle box refers to the event stream, source, or feeds. The bigger three-dimensional box refers to CEED has several components such as Input adapter, Output adapter, CEP-core, and CEP libraries. In addition, this embedded device holds some system libraries that are required for the functioning of the embedded device. Any devices, any event source, or feeds, can be connected to CEP engine as input event source or output event receiver. The CEP engine receives or sends the events in any format such as digital data, analog data, or serial.

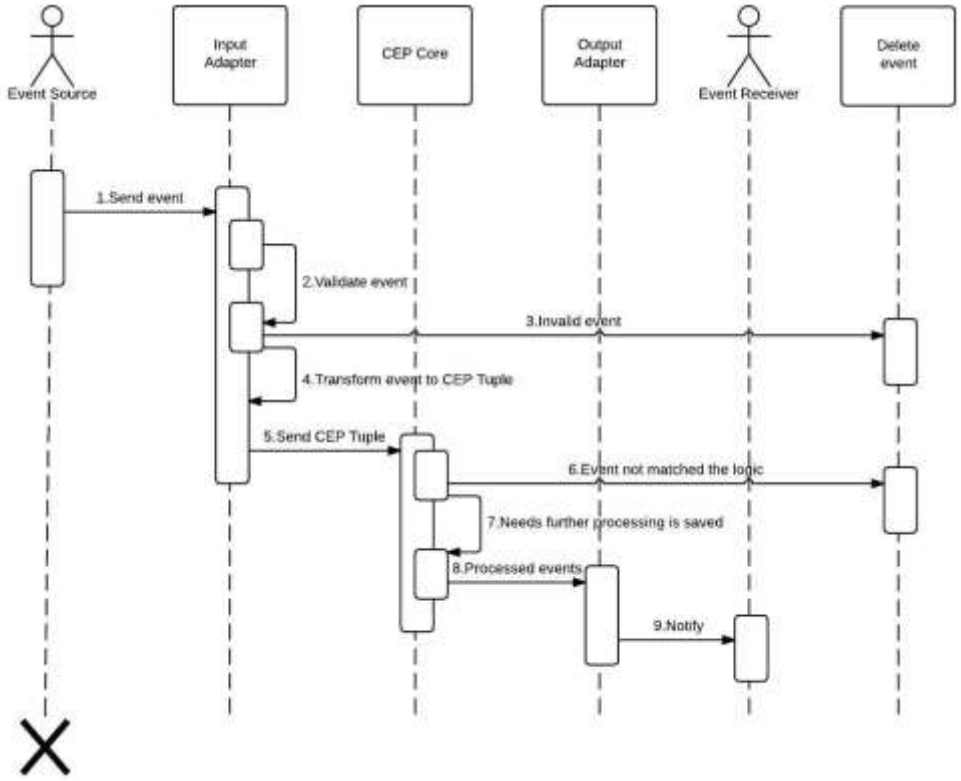

Figure 3.6: Sequence Diagram of proposed CEED.

3.3 Design Considerations

3.3.1 Usage flow for proposed CEED

As the memory and CPU capacity of embedded devices are very limited, any program designed for these devices must be of small size and lightweight. Lightweight program means, the program should not perform any heavy lifting such as performing complex duties. It should accomplish only a minimum number of tasks. To fulfill the characteristics of the embedded device programming, during the CEP design following design decisions were made:

- To reduce program size, it was decided to support the model, which the program loaded to Arduino/ Embedded device. It should be optimized and support only a custom query.

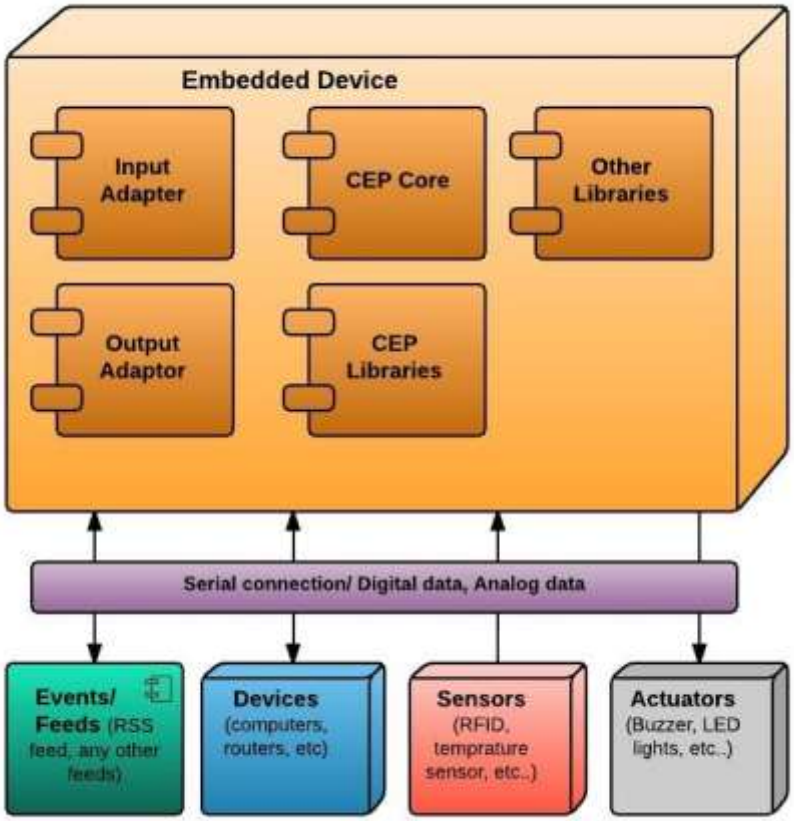

Figure 3.7: Deployment view of the proposed CEED.

- To reduce the extra unnecessary processing such as creating a tree for the query and conducting parsing dynamically, the query processing was moved away from the Arduino program. This action further reduced program size.

These decisions forced to adopt the usage flow described in Figure 3.8 for the proposed CEP engine. First, define the events stream, query, and Input/Output adapter settings using web site. Web site will generate the Settings.xml, automated script, desktop program, and generate the ZIP package with required CEP libraries in addition to the above two files.

Automated script starts the desktop program, which validates the query and other info in the Settings.xml, and any errors found will be notified. If Settings.xml is valid, then desktop program constructs the parse tree using the query. Next step is to

create the sketch for the Arduino using this parse tree and other info from the *Settings.xml*.

Desktop application to generate Arduino sketch from the Settings.xml is in Java. This application is in the jar file format that accepts the Settings.xml and parses it using XML parser. Using the ANTLR, the CEP query is validated next. If the query is valid, the application will create a parse tree for that query. Finally, by using the depth-first traversal, the parse tree is traversed to generate the Arduino sketch.

In order to reduce the complexity of the auto-generating sketch, there were few common libraries written, which consists of common methods and provide support for methods on this auto-generated sketch. Arduino sketch could directly use this since these libraries were written in C++ library format.

The generated sketch is compiled and uploaded to the Arduino board by automated script. Arduino will start functioning as soon as the program upload completes. Once the sketch uploads, the user needs to connect event sources and event receivers to the Arduino board. Upon the completion of the entire setup, user is safe to press the reset button in Arduino, which restarts the program and get ready to receive events from event source/s. In addition, entire process is presented in Figure 3.4 as well.

In proposed CEP engine, the query cannot be dynamically modified. Dynamically modifying CEP query means the ability to assign or modify the query without restarting the device.

3.3.2 CEP Tuple

As shown in Figure 3.9, CEP engine represents events using a tuple data structure similar to Siddhi [9]. Each CEP Tuple holds a single event data in the CEP memory.

CEP tuple was initially designed with the following format:

- *ID:* Unique ID within the stream and the data type of this variable is long
- *Time Stamp:* Time from the start of the Arduino board in milliseconds and its data type is long

- *Data 1.*, Data N: These are the individual value units, which are stored in String Array

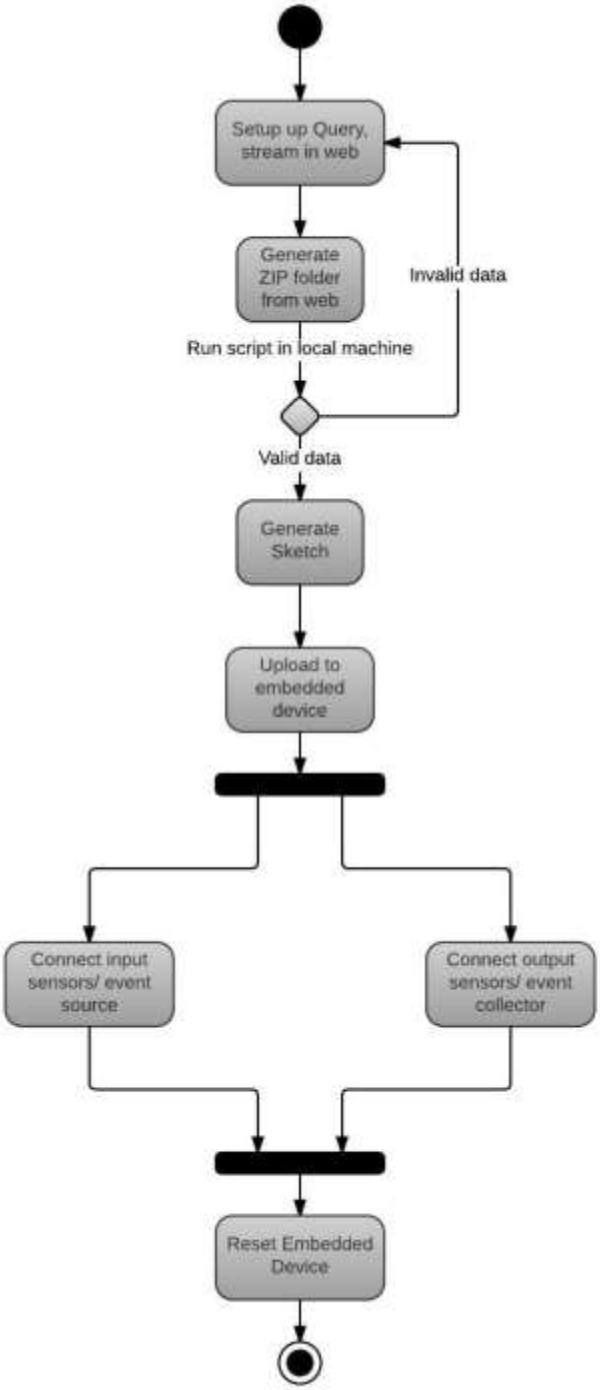

Figure 3.8: Usage flow for the proposed CEED.

Figure 3.9: Proposed CEP Tuple structure.

The Tuple design was considered unsuitable after analyzing the performance and memory profile of the initial implementation. Main reason for the failure was because the String Array requires the pointer usage but Arduino is not efficient in freeing up this mass memory. Thus, each event leaves some non-reclaimable memory, which soon filled up the entire memory of Arduino.

To overcome this memory leak, a single String object was used instead of having the String array to hold data. Each unit in the String is separated by '|' character. Using single String slows down the processing in comparison to the Array implementation, since it requires costly String manipulation actions to add, remove, and modify a data in a tuple. Anyway, this helps to remove the usage of pointers so the memory reallocation normally takes place. In addition, using single string uses less memory than using the String array to hold the same data. Hence, single String object is selected to hold the entire data of the tuple instead of the String array, because removing memory leak and reducing memory usage is more important than the performance, with respect to embedded devices.

It is vital to optimize each byte whenever possible since the basic Arduino memory is limited to 2KB. In CEP engine library, the CEP Tuples are stored in linked list, so the order is preserved. In addition, it ensures that unique ID is never used in any of the common library functions that makes this unique ID as optional. Therefore, the unique ID was removed from the CEP tuple design and the design illustrated in Figure 3.9 was adopted as the CEP Tuple structure.

3.3.3 Single processor model

The Arduino board and most of the embedded devices support single processor or single thread model. Therefore, CEP engine was forced to select the Single processor model than selecting the pipeline architecture similar to Siddhi [9].

3.3.4 CEP Engine Libraries

CEP has six C++ libraries in total, to support the main sketch that will be generated by the desktop tool. The libraries were divided in such a way that ensures each sketch do not require to bind with all the supporting libraries for the in proposed CEP engine. The sketch only bound with the required libraries, made the size of the program to reduce drastically.

Libraries implemented for CEP engine were as follows:

- CEPData: have a single C structure only, which is the implementation of a CEP Tuple
- CEPPattern: have helper methods for the Pattern query type
- CEPSequence: have helper methods for the Sequence query type
- CEPStream: have all methods required for event stream
- CEPUtility: have helper methods for manipulating String data type
- CEPWindowUtil: have helper methods for window type queries

For example, if we consider the CEP engine expects to run a query, which is a Pattern type, then the sketch will include only CEPData, CEPPattern, and CEPUtility libraries. If the type of query is Filter, then Arduino sketch will include only CEPData and CEPStream libraries.

3.3.5 State machine

Similar to Siddhi, state machine supports to implement the Pattern and Sequence type of queries [9].

Pattern

Pattern queries fires an event if the series of conditions get satisfied one after the other. Consider the example where we consider the pattern as A->B->C.

Consider the incoming event sequence as follows:

A1, A2, B1, A3, A4, B2, C1, C2, A5, B3, C3, B4

Here, the first event is fired when the system receives C1, where the captured pattern is A1, B1, and C1. The second event will fire when system receives C3, where the captured pattern is A5, B3, and C3.

Sequence

Sequence queries fires an event if the series of conditions get satisfied one after the other consecutively. Consider the example where we consider the sequence as A -> B -> C. Let us consider the incoming event sequence as follows:

A1, A2, B1, A3, A4, B2, C1, C2, A5, B3, C3, B4

Here the only event fired when the system receives C3, where the captured sequence was A5, B3, and C3.

3.3.6 CEP query language specification [16]

Following provides an abstract BNF based definition for Siddhi language.

```
<execution-plan> ::= <define-stream> | <execution-query>

<define-stream> ::= define stream <stream-name> <attribute-
name> <type> {<attribute-name> <type>}

<execution-query>::= <input> <output> [<projection>]

<input> ::= from <streams>

<output> ::= ((insert [<output-type>] into <stream-name>) |
(return [<output-type>]))

<streams> ::= <stream>[#<window>]

          |   <stream>#<window> <join> <stream>#<window> on
      <condition> within <time>

          |   <stream> -> <stream> ... <stream> within <time>
```

```
          |  <stream>, <stream>, <stream> within <time>

<stream> ::= <stream-name> <condition-list>

<projection> ::= (<external-call> <attributelist>) |
<attributelist> [group by <attribute-name> ][having
<condition>]

<external-call> ::= call <name> ( <param-list> )

<condition-list> ::= {'['<condition>']'}

<attributelist>::=(<attribute-name> [as <reference-name>]) | (
<function>(<param-list>) as <reference-name>)

<output-type> ::= expired-events | current-events | all-events

<param-list> ::= {<expression>}

<condition> ::= ( <condition> (and|or) <condition> ) | (not
<condition>) | ( <expression> (==|!=|>=|<=|>|<) <expression> )

<expression> ::= ( <expression> (+ | - | / | * | %)
<expression> ) | <attribute-name> | <int> | <long> | <double>
| <float> | <string> | <time>

<time> ::= [<int>( years | year )]  [<int>( months | month
)] [<int>( weeks | week )] [<int>( days | day )] [<int>( hours
| hour )] [<int>( minutes | min | minute )][<int>( seconds |
second | sec )] [<int>( milliseconds | millisecond )]
```

The query language specification for the CEP engine is entirely based on the WSO2 Complex Event Processor 3.1.0, where the WS02 Complex Event Processor is based on Siddhi. In other words, CEP query language is the simplified version of WSO2 Complex Event Processor 3.1.0 query language.

Event Stream definitions

All streams that cannot be derived from queries must be defined before the use. Event stream definition can be define as following,

```
<define-stream> ::= define stream <stream-name>
<attribute-name> <type> {<attribute-name> <type>}
```

Example stream definition as follows,

```
define stream RoomClimate (temp float, humidity int);
```

Pass-through

```
from <stream-name>
select ( {<attribute-name>}| '*'|)
insert into <stream-name>
```

Pass-through query creates an output stream according to the projection defined and inserts any events from the input stream to the output stream. Projections can be either

- All (*)
- Selected attributes.

Filter

```
from <stream-name> {<conditions>}
select ( {<attribute-name>}| '*'|)
insert into <stream-name>
```

Filter query creates an output stream and inserts any events from the input stream that satisfies the conditions defined with the filters to the output stream. Filters support following types of conditions,

1. >, <, ==, >=, <=, !=
2. and, or, not

Example filter as follows,

```
from RoomClimate[temp >= 20 and temp < 25]
select temp,humidity
insert into IdealRoomClimate
```

Windows

```
from <stream-name> {<conditions>}#window.<window-
name>(<parameters>)
select ( {<attribute-name>} | '*' |)
insert [<output-type>] into <stream-name>
```

Window is a limited subset of events from an event stream. Users can define a window and then use the events on the window for calculations. A window has two types of output that are current events and expired events. A window emits current events when a new event arrives. Expired events are emitted whenever an existing event has expired from a window.

CEP engine supports only Length Window and Time window. In addition, CEP engine supports the following type of aggregate functions such as sum, average, max, min, and count.

Joins

```
from <stream>#<window> [unidirectional]
      join <stream>#<window> [unidirectional]
[on <condition>] [within <time>]
select ( {<attribute-name>}| '*')
insert [<output-type>] into <stream-name>
```

1. Join takes two streams as the input
2. Each stream must have an associated window
3. It generates the output events composed of one event from each stream
4. With "on <condition>" Siddhi joins only the events that matches the condition
5. With "within <time>", Siddhi joins only the events that are within that time of each other

CEP engine supports only Join or inner join.

Pattern

```
from <stream> -> <stream> ... <stream> within <time>
select <attribute-name> {,<attribute-name>}
insert into <stream-name> partition by <partition-id>
```

1. Pattern processing is based on one or more input streams.
2. Pattern matches events or conditions about events from input streams against a series of happen before/after relationships.
3. The input event streams of the query should be referenced in order to uniquely identify events of those streams. e1=Stream1[prize >= 20] is an example of a reference.
4. Any event in the output stream is a collection of events received from the input streams, and they satisfy the specified pattern.
5. For a pattern, the output attribute should be named using the 'as' keyword, and it will be used as the output attribute name in the output stream.

If "within <time>" is used, CEP engine triggers only the patterns where the first and the last events constituting to the pattern have arrived within the given time period.

Can combine streams in patterns using logical OR and AND.

- and - occurrence of two events in any order

- or - occurrence of an event from either of the steams in any order

Can count the number of event occurrences of the same event stream with the minimum and maximum limits. For example, <1:4> means 1 to 4 events, <2:> means 2 or more, and [3] means exactly 3 events.

Sequence

```
from <event-regular-expression-of-streams> within <time>
select <attribute-name> {, <attribute-name>}
insert into <stream-name>
```

With patterns, there can be other events in between the events that match the pattern condition. In contrast, sequences must exactly match the sequence of events without any other events in between.

1. Sequence processing uses one or more streams.
2. As input, it takes a sequence of conditions defined in a simple regular expression fashion.
3. The events of the input streams should be assigned names in order to uniquely identify these events when constructing the query projection.
4. It generates the output event stream such that any event in the output stream is a collection of events arrived from the input streams that exactly matches the order defined in the sequence.
5. For a sequence, the output attribute must be named using the 'as' keyword, and it will be used as the output attribute name.

When "within <time>" is used, just like with patterns, CEP engine will output only the events that are within that time of each other.

Following regular expressions are supported:

- Zero or more matches (reluctant).
- + One or more matches (reluctant).
- ? Zero or one match (reluctant).

3.5 Comparison of Siddhi and CEP Engine for embedded devices

This section discuss about the main differences between Siddhi and CEED. Table 3.1 compares between Siddhi and the CEP engine for embedded devices. This comparison is important, since CEP Engine for the embedded devices is based on Siddhi and uses Siddhi query language. Thus, the engine will have several similarities and differences, because the operation use-cases and target devices are completely different.

Both are CEP engines while Siddhi is targeted for the servers/ desktop computers where minimum configuration should satisfy 2GB RAM and 1 GB hard disk space. The CEP Engine for embedded devices targets for edge devices and the minimum configuration is based on Arduino UNO (2KB RAM and 32KB storage). Both engine represents events using a tuple data structure but slightly different on the structure. Siddhi supports duplicate event detection while CEP Engine for embedded devices does not support this feature.

Table 3.1: Comparison of Siddhi, and CEED.

	Siddhi	**CEED**
Minimum system requirement	Desktop machine with 2GB RAM and 1 GB storage space	Arduino UNO (2KB RAM and 32 KB storage)
Architecture	Uses Pipeline model with multi-threaded execution	Single threaded execution
Events	CEP Tuple	CEP Tuple
	Duplicate event detection	No duplicate event detection
Query Language	Supports advanced queries including sub queries and different type of queries in a single query (e.g., Filter type and Pattern type in a single query)	Not support advanced queries: only one type of query is supported
	Supports Siddhi query language	Based on Siddhi query language with limited support
	Each CEP engine deployment can be modified with different queries on the go	Each CEP Engine deployment binds with the single CEP query. Changing the query requires new deployment.

Siddhi uses the pipeline model to execute query and it uses multi thread/ multi-processor based execution. Arduino board and a majority of embedded devices

supports single processor or single thread model forced the CEP Engine for embedded devices to select the single processor model, unlike Siddhi.

Similar to Siddhi, the CEP engine for embedded devices use state machine to implement the Pattern and Sequence features. CEP Engine for embedded devices uses the same Siddhi query model as query language but with limited functionality. CEP engine for embedded devices is not supporting either Tables or Partitions in query like Siddhi. Filter type does not support the 'contains' and 'instanceof' keywords, Window type supports only Length window and Time window, Joins type supports only two streams that can be joined and support only inner join, and Pattern type does not support 'every' keyword. The CEP engine for embedded devices does not support advanced query or sub query type but only supports single type in a query.

Chapter 4

DEVELOPMENT

This chapter discusses implementation details of the CEP Engine for embedded devices. Initially it explains about the software process followed to manage this project then move on to discuss other details such as standards followed, tools used to track project, and version controlling.

4.1 Software Process

As the CEP engine is to be used in embedded devices, its design and implementation is constrained on the limited memory and lower computing power available in devices. Due to these constraints, as well as there were no proven reference architectures for CEP implementations in embedded device, it was difficult to follow a concrete software design process at once. Owing to the above reasons, two software process models, combined in such a way to benefit this type of product nature were adopted. They were based on Software Prototyping and Agile Process.

Software prototype model is the activity of creating prototype of software components/applications such as incomplete version of the software program, which is constructed under these circumstances. This prototype typically simulates only a few aspects of, and may be different, from the final product. It was decided therefore, to prototype more important components that will have a direct effect on the program size, the memory uses, the load to the CPU, and the processing time. It was chosen to implement the CEP Tuple and Common library, which are the two main components influencing above parameters. Therefore, after each prototype, conducting a performance analysis was required to find the dynamic memory usage, execution time, and program size. Depending on the analysis results, the alternative was to redesign, implement the prototype, and analyze, then repeat the above steps until the design and analysis results were satisfactory.

Due to the above level of agility and the prototyping, it was decided to use the similar framework called Dynamic Systems Development Method (DSDM) [39]. As shown in Figure 1, after the project idea the main components were identified and create prototypes based on framework similar to DSDM (in light blue box). DSDM is an Agile project delivery framework, primarily used as a software development method. It is an iterative and incremental approach that embraces the principle of Agile development, including continuous user involvement [39]. This project used the mixture of throwaway prototypes and evolutionary prototypes to understand and evolve the system during early stages of the project. DSDM defines four categories of the prototyping, where Capability/Technique Prototype, and Performance and Capacity Prototypes are the two prototyping categories used in this project lifecycle. This approach assisted extremely well in identifying the performance and memory related problems early in the project, to reduce risks at later development.

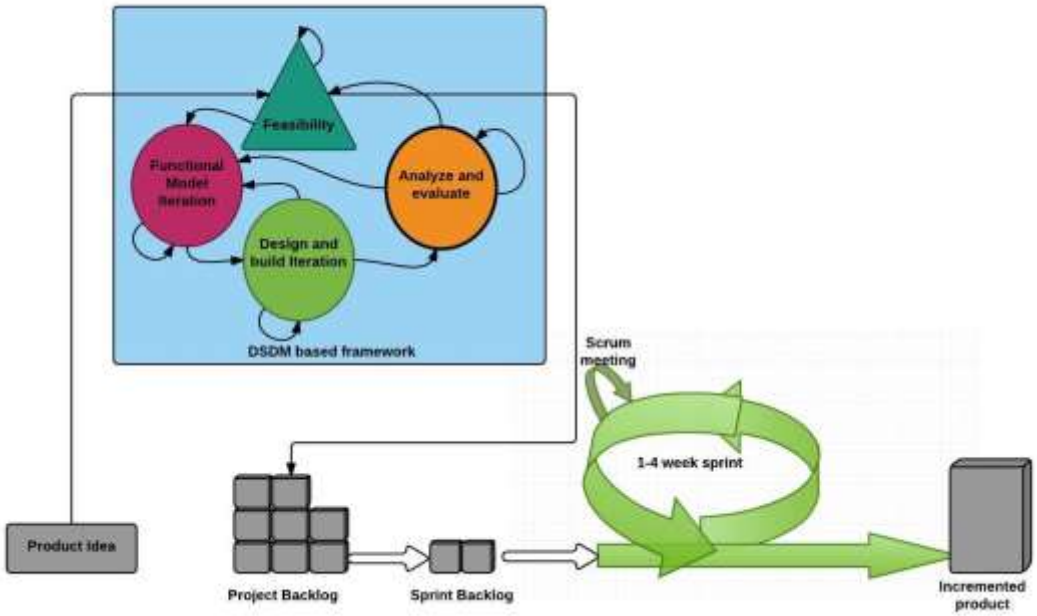

Figure 4.1: Process for the CEP engine for embedded devices.

During later stages of product implementation, the Scrum was selected as a process. As Figure 4.1 illustrates, after the DSDM the project backlog will be completed and several items from backlog chosen for each Scrum. Scrum is an iterative and incremental Agile software development methodology for managing product

development. Sprint (iteration) is the basic unit of development in scrum. The duration is fixed in advance for each sprint and is normally between one week and one month, two weeks being the most common. Each sprint starts with a sprint-planning event, the aim of which is to define a sprint backlog, where the work for the sprint is identified and an estimated commitment for the sprint goal is made.

In addition to the product development, writing the thesis is also benefited from scrum methodology. Since this is an individual project, the daily scrum was not followed. However, an active communication with my supervisors helped to make any timely decision during the sprint. Each sprint was decided after discussions with the supervisor regarding the tasks to be included and the duration at the beginning of each sprint. The scrum methodology greatly assisted to plan the tasks in detail for each sprint, to obtain regular and timely feedback from the supervisors, and to continue the time plan to make this project a success.

4.2 Coding Standards and Best Practices

A set of coding standards and best practices to maintain the standard and readability of the source files was followed.

Comments

All classes and functions must have comments. Class comments should explain the purpose of the class, method comments explain the parameter, return type, and method related info. In case of any complex logic, the line comments were added in appropriate places within the functions.

Readability

Make sure to add enough comments with descriptions to make the code well readable. Define the classes and packages in a meaningful way to make sure all related files are found in a single place. Always use meaningful self-descriptive names to classes, methods, and variables to make the code more readable.

Remove unnecessary commented blocks, and unused code segments from source files to ensure a tidy appearance. Always use the code with correct indentation.

Committing to the repository

Always add descriptive and meaningful comments with each commit. Use branches when necessary such as fixing bugs while the new feature is on development. Make sure the master branch always have the runnable code.

Other coding guidelines

- Each method should serve single functionality and do not clutter with multiple functionality
- Whenever possible, return the status at the end of the method function
- Do not use too many parameters as a method signature
- When returning array or collection, use empty array or collection instead of null/NULL if there are no data to return
- Add line comments at the end of each block to identify the ending block when there are multiple blocks ending.
- Always use braces to surround the code block, even if it is the single line IF conditions.

Java Specific Standards

Google style guide for Java programing is used [40].

FindBugs program search bugs in Java programs. Eclipse plugin version of FindBugs used to analyze java code used in this project. http://findbugs.sourceforge.net/

Arduino Specific Standards

The Arduino style guide for creating CEP libraries, which is recommended by Arduino main website [41].

4.3 Project Management and Tracking

OneNote is a planner and a note taking software from Microsoft. My supervisor Dr. Dilum Bandara introduced this as a collaboration tool between student and supervisor, and it is used effectively as the Project Journal throughout the project duration.

Following are many ways OneNote was employed throughout the project:

- to capture project related interesting material at one place in any format, and store and share files
- to use as a first draft for project documentation
- as a planner and tracking tool
- accessible anywhere, even in mobile

4.4 Version Control

In software engineering, version control is any kind of practice that tracks and provides control over changes to source code. Software developers occasionally use version control system to maintain documentation and configuration files in addition to the source code. Therefore, it is essential to have a version control system to almost all software projects.

Git is the software that runs at the heart of GitHub [42]. Git is version control software, which means it manages changes to a project without overwriting any part of that project [42]. GitHub is a web-based Git repository hosting service, which offers all the distributed revision control and source code management functionality of Git. In addition, GitHub adds its own features [42].

GitHub makes Git easier to use in two ways: First, GitHub software, which can install to any computer, provides a visual interface to help and manage version-controlled projects locally. Secondly, creating a project in GitHub brings the version-controlled projects to Web, enable to collaborate, and it provides additional features such as wikis and basic task management tools for each project in GitHub.

CEP engine for the embedded devices product will be Apache licensed open source product so the source code and everything related to this project will be on web for anyone to access, contribute, or improve this product. Hence, it requires a proper version control system to collaborate with anyone using the website actively and GitHub fits perfectly for this goal.

Furthermore, the wiki and other features offered by the GitHub provide an easy route to create help documents, manuals, and site for the product. More importantly, this is a free service and popular for the open source projects management. Therefore, all the above-mentioned reasons compelled to select GitHub as the versioning control system for this project.

Chapter 5

PERFORMANCE ANALYSIS

This chapter presents the performance analysis of CEP Engine for embedded devices. As Arduino UNO is the basic and popular board among the other Arduino boards, it was used here to obtain all the data presented.

5.1 General Assumptions and Guidelines for the Analysis

The Input adapter and Output adapter implementations highly depends on the format that each event receives and the format each event (i.e., direct reading of the pin, the string with some separator, and certain other formats such as XML) to be send out from the CEP Engine for embedded devices.

In addition, the time spend in Input/Output Adapter is not related to type of query the CEP engine is running. Hence, the time duration spent in the Input Adapter and Output Adapter for the time duration analysis was not included. However, in the memory analysis, the memory usage during the Input/Output adaptor was included in order to understand the dynamic memory requirement to run the program during program execution.

In addition, we will look into the time spent for the whole process between the time duration from reading the event of the sensor/source to writing back the event to the sensor/event receiver in Section 5.4. This duration will assist to form an idea about the throughput of CEP engine for embedded devices.

As illustrated in Figure 5.1, the time duration analysis we measure in this analysis is the time taken in the core process (marked in the figure) for each type of the query, the CEP engine is running.

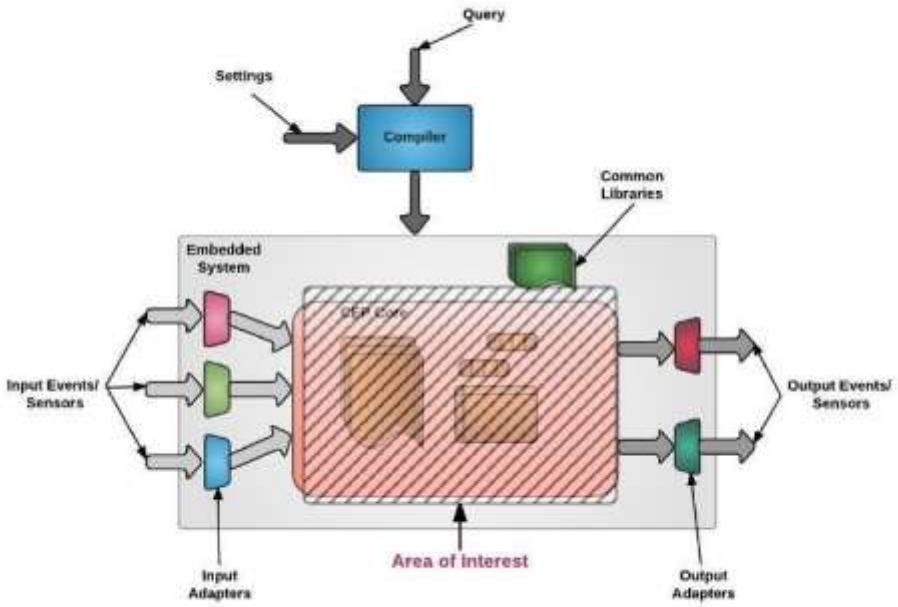

Figure 5.1: Area of Interest: location the time analysis.

Even though CEP Engine for embedded devices support six types of queries including Pass through, Filter, Window, JOINs, Pattern, and Sequence, we will only look into three types of queries as follows for the analysis purpose:

- *Filter*: The Pass through and Filter types are very similar as it is the subset of events or subset of event parameters as output. Hence, for this analysis, we have chosen the Filter type for analysis.
- *Window*: The time window, length window, and JOINs are all depends on the windows. Hence, to get the idea, we chose length window for the analysis.
- *Pattern*: The Pattern and Sequence are built on state machine type and we have selected pattern query for the analysis, as it is the state machine type query.

Two streams used for the analysis are as follows:

- `define stream temperature (humidity float, temp float);`
- `define stream lightlevel (level float);`

The queries used for the analysis is defined as follows:

- Filter Query

```
from temperature [temp>24 and temp <=27]
select id, temp
insert into roomtemperature;
```

- Windows Query

```
from temperature[temp >= 30]#window.length(5)
select temp, avg(humidity) as avgHumid
having avgHumid>90
insert into roomtemperature for expired-events;
```

- Pattern Query

```
from e1=temperature[temp >= 40] -> e2=temperature[temp >=
50]<3:> -> e3=lightlevel[level < 400]
within 5 min
select e1.temp as temp, e3.level as light
insert into fire;
```

Figure 5.2 shows the setup of fire alarm simulation circuit, which includes Temperature and Humidity sensor (AM 2301), Photocell sensor, Piezo Buzzer, LED light, wires, resistors, breadboard, and Arduino UNO. The above setup is used to test the patterns query.

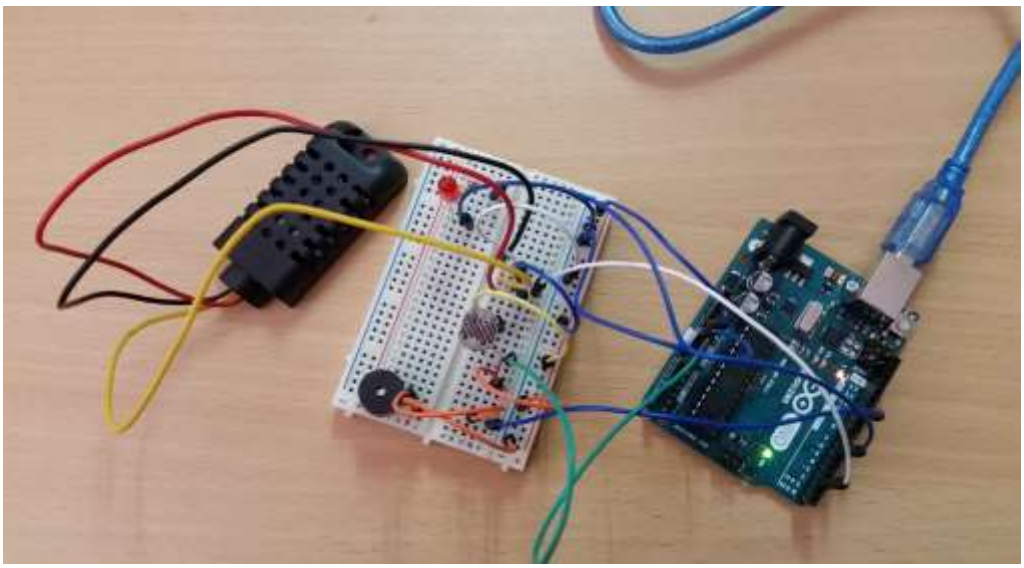

Figure 5.2: Fire alarm setup used to for the pattern type query analysis.

5.2 Query Processing Time Analysis

The time duration used for the main process for each feature type is one of the important parameters to get to know by the users. It gives the idea of how much time the Arduino board will occupy to process each type of query. CEP Tuple with two data/value units are used throughout the analysis.

Filter

Figure 5.3 illustrates, the duration to process the filter query for 25. Event up to #6 does not satisfy the filter condition, and remaining events satisfied the filter condition. This shows that the events that does not satisfies the filter conditions spends ~220 microseconds (~0.2 millisecond) in CEP Core, while the events that satisfy the filter condition spends ~1,112 microseconds (~1.2 millisecond).

Because Filter type query does not involve in storing or using previous data for the processing, the duration taken to process the filter query is always similar. Events that do not satisfy the filter condition will be discarded soon after it determined, and that is the reason unsatisfied events takes very little time in the filter function.

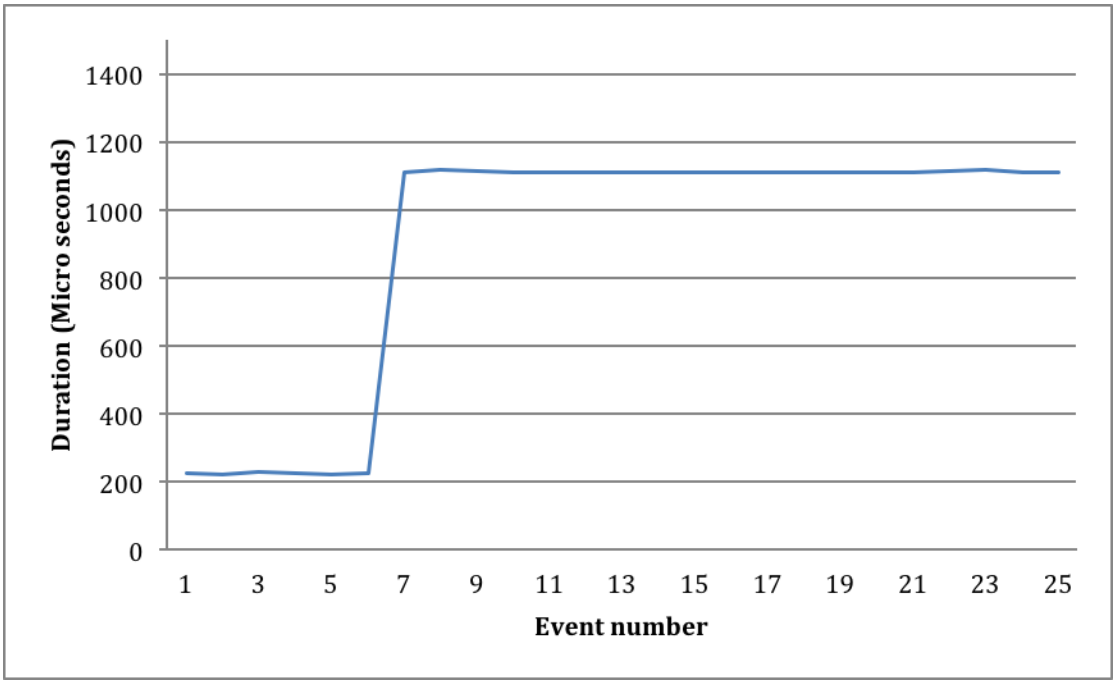

Figure 5.3: Duration to process the Filter query for 25 runs.

Window

Figure 5.4 illustrates the duration to process the window query (for 25 runs) each with three different window sizes including window size 5, 10, and 20. Events up to #5 were inserted into the windows; however, they did not fill the windows entirely. Because of that there were no further processing other than just inserting the event to the window. This takes ~900 microseconds (~0.9 millisecond) for all three-window sizes. Events #6 to #9 are the events that arrive after the windows is full but the expired event does not match the given condition. This takes ~1,450 micro seconds (~1.5 milliseconds) for all three window sizes.

According to the design, expired event is always the first event in the linked list regardless of what the window size is. Hence, the time taken to remove the CEP tuple from the window, and determine if the event matches the condition will be same regardless of the window size.

The events from #10 to # 25 arrive while the window is full and expired events due to this event satisfy the having condition as well. Window size 5 and window size 10, requires ~2,600 microseconds (~2.6 milliseconds) respectively, while window size 15 require ~2,750 microseconds (~2.8 milliseconds).

The CEP implementation logic should behave same theoretically, as there were no reason for the time taken will increase with the window size. However, the CEP library requires several operations with the third-party Linked-list library in backend. So the third-party linked list implementation getting slower when the number of elements it holds increases. This slight increase in time can be noticed between window size 5 and window size 10. This difference is seen clearly between window size 10 and window size 15.

Figure 5.4: Duration to process the Window query (25 runs each) for three different window sizes.

Pattern

Figure 5.5 illustrates, the duration to process the Pattern query for 25 runs. Up to event #5, the events that does not match the pattern, which is discarded in CEP Core. Time duration for processing non-matched events is ~650 microseconds (~0.6 milliseconds). Event #6 to event #10 is matching events, which were inserted to form the pattern (not the event which completes the pattern), takes the duration of ~2,300 microseconds (~2.3 milliseconds). The events from #11 to #25 are the events, which completes the pattern, takes the duration of ~4,600 microseconds (~4.6 milliseconds).

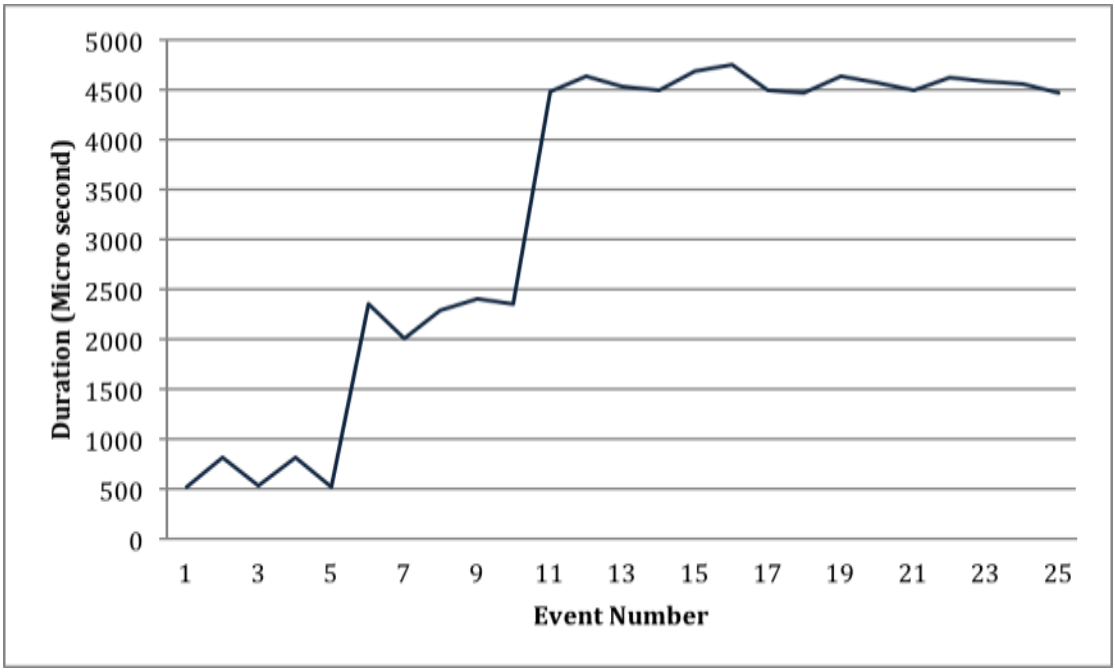

Figure 5.5: Duration to process the Pattern query (for 25 runs).

5.3 Memory Analysis

To observe the usage of memory increase during the process, there were three readings used to understand the memory usage at each step.

- The memory usage reported after the initialization of streams and other methods.
- The memory used after 25 events.
- Memory used when the event is writing the value to the desired sensor/output source. For this analysis, we used this value because the highest dynamic memory usage in each event processing is just before the event sent out to sensor or event receiver.

These data will help to identify how much extra memory required for the process to determine the size of the windows, JOINs, Pattern events, and Sequence events.

```
int freeRAM() {
  extern int __heap_start, *__brkval;
  int v;
  return (int) &v - (__brkval == 0 ? (int) &__heap_start :
(int) __brkval);
}
```

The above code segment is used to calculate the free RAM for memory analysis. The dynamic memory of the Arduino UNO is 2,048 bytes (2 MB).

Table 5.1 compares the memory usage in different stages of processing in the Arduino UNO board. This gives the idea of how the memory requirement differentiate depend on the query type. For the window query type analysis we have choose window with size 15 for this analysis.

Table 5.1: Memory analysis for the query types.

	Filter (Bytes)	**Window (Bytes) Size =15**	**Pattern (Bytes)**
Memory used after initialization	511	492	1,090
Memory used after 25 events	511	1,570	1,225
Memory used just before writing output event	760	1,854	1,225
Dynamic memory required for the process:	250	280	54
Free memory available when the memory usage is high	1,288	194	769

5.4 General performance analysis

To obtain the idea of throughput, it is required to understand the total time taken for a single CEP event to be read from the sensor/source to that event to be written to the Sensor/Event receiver. For this analysis, the CEP Tuple with two data/value units are used and we used the filter query we used in this section for this analysis.

In addition, there were five readings recorded to understand the time taken in each step, which are as follows:

- During reading the value from the sensor – ~250 microseconds (~0.3 millisecond for serial and for digital reading it is smaller as ~10 microseconds)
- During the input adaptor process - 900 microseconds (~0.9 millisecond)
- During the CEP Core process – 1,120 microseconds (~1.1 milliseconds)
- During the output adaptor process – 560 microseconds (~0.6 milliseconds)
- During writing the value to the sensor – 300 microseconds (~0.3 milliseconds)
- Total duration for one filter event to process – 3,130 micro seconds (~3.1 milliseconds)

Moreover, we noticed the duration for a method call is ~60 microseconds. We obtained an estimate of the memory usage of a single CEP Tuple, which consists of two data/value units is ~54 bytes.

5.5 Summary

Following are some of the important values or data, which are valuable to use the CEP Engine for embedded devices effectively. The below data will help in planning the use case that to be used with the CEP Engine for embedded devices.

- Keep ~300 bytes free for the dynamic memory requirement of any type of query processing, when planning for the window size or pattern or selecting the suitable board.
- It is better to allocate ~60 bytes when calculating the memory for each CEP Tuple when planning.
- For the UNO board it is advised to not go beyond 15 elements in window.
- The Arduino UNO board can process ~300 events per second (throughput) when considering the filter query defined in Section 5.1.

However, it is important to note that the throughput is highly depends on the input sensor reading/writing and Input/Output adaptor.

Chapter 6

SUMMARY AND FUTURE WORK

Issues identified in CEP Engine for embedded devices are discussed in Section 6.1. Limitations are presented in Section 6.2. Section 6.3 briefly discusses the work performed in this project, including the development of CEP engine for embedded devices. Finally, this thesis continues describing the identified future work.

6.1 Conclusion

With the popularity of the IoT, sensors were placed everywhere. These sensors generate continuous streams of data, and in many cases, those streams need to process in near real time. To process these continuous streams in real time, a new form of data processing called Complex Event Processing (CEP) was introduced. Work conducted here proposes to push this CEP engine to the embedded device that lives closer to these sensors, rather having this in the powerful servers or in cloud. Therefore, many simple decisions will be made locally and the volume of data transferred through network to the traditional CEP Engines will reduce drastically. This enables rapid responses to detected events and free up the network bandwidth considerably.

We developed a CEP engine for resource constrained embedded devices to be placed the edge of the IoT network. The CEP engine is developed for Arduino, as it is a globally popular, open source hardware platform with a massive community base. In addition, our CEP engine for embedded devices uses Siddhi Query Language, which is similar to SQL queries. This enable rapid development of IoT applications as the CEP capabilities can be added to embedded devices just by writing an SQL-like query. This CEP engine adopts single threaded model because majority of the embedded devices including Arduino are single threaded. Another major decision made was to create the CEP Engine for predefined query, rather than support for

dynamic query assignment like Siddhi. Reason for this decision was to reduce memory usage and CPU usage of the embedded engine. Furthermore, CEP Engine uses the state machine to implement the Pattern and Sequence type queries and uses tuple-type data structure for internal processing. In addition, it supports Pass through, Filter, Window, and JOINs type query as well.

Several data were captured and analyzed while the CEP Engine processed several basic queries from an experimental setup. Performance analysis provides an overall idea of speed of query execution, memory usage, and tuple sizes, which is important in selecting the required embedded system board for each use case. Moreover, the performance analysis demonstrated that even with a low-end Arduino board a large number of events could be processed in real-time while having a lower impact of processor and memory utilization.

In conclusion, the data we discussed in this thesis proved the possibility of effectively using CEP engine in the embedded devices and solving our target problem we wanted to address. This work is an ongoing effort and will be distributed under the Apache License, once this proof of concept work reaches closer to the end product. Therefore, we have identified the limitations of current CEP Engine for embedded devices, and the future work, and working towards to make this as a powerful end product.

6.2 Known Issues

CEP Engine for embedded devices is the ongoing project, which is being built until it is stable and ready for production usage. Even though the “time window” does not support expired event manipulation, CEP Engine will never alert user for unsupported feature, if the time window query includes the expired event processing. Therefore, only testing can identify such issues.

6.3 Limitations

CEP Engine for embedded devices is the proof of concept to describe the absolute possibility of the CEP Engine for embedded devices efficiently. Hence, this product needs to address few limitations and through quality, assurance before applying for any complex deployments unless reviewed the code thoroughly before usage in such instances.

Time window is not supported expired event manipulation

The events processed in windows type queries are based on three event-processing scenarios, which are:

- Current event can be processed
- Expired event can be processed
- Both current and expired events can be processed

"Length window" supports all three types above while "Time window" only supports the current event. Reason for this limitation is that the current CEP engine for embedded device does not have any watchdog to be triggered in the event of any time window event expires; however, the time window will first remove all the expired events upon receiving a new event before processing the current event.

In "Length window", an event will only expire upon receiving of a new event. Thus, no difficulty will arise in supporting all three event-processing scenarios. However, in "Time window", an event can expire at any moment so this requires some watchdog to support the expired event-processing scenario.

Requirement to have basic knowledge on Arduino program to use CEP Engine in complex scenarios

The Arduino sketch is generated based on the user-defined parameters, query, and stream definition arriving from web site. Anyway, the web site is not well designed to capture most of the complex scenarios of formatting the Input/Output adaptors. The web site only supports basic sensor reading and writing from/to PINs and partial Serial Input/Output.

Therefore, the user of the system required knowing basic Arduino programming to define the logic of Input and Output adaptors in complex scenarios. This is one of the major limitations for this product, hindering the wide and efficient use by the users.

Lack of Error handling

The errors identified in the Siddhi Queries are not handled properly to give the clear meaningful error messages to the user consisting why, what, and how the error needs to be rectified. This proper error handling is one of the main requirements before it is used in the production environment as an end product.

Possibility of missing events

There is a possibility of missing events due to pooling and time taken to process events. This proposed CEP engine does not explicitly take care of pooling all the events before entering to the CEP engine. Due to this limitation, there are possibilities of missing some events when the time taken to process an event is much higher than the time interval between each event arriving to CEP because the proposed CEP engine is single threaded.

6.4 Future Work

We planned to extend our work on following directions.

Implement the support for missing keywords of Siddhi Query Language for the supported features

CEP Engine for embedded devices support six features, which are the Pass through, Filter, Windows, JOINs, Pattern, and Sequence. Though it supported the above features, CEP Engine possess several limitations as missing the support for some keywords of the Siddhi Query Language that are related to the above six features, described in Section 2.5.

Following is a list of features that could be implemented:

- Support ‘contains’ and ‘instanceof’ keywords in the Filter type queries

- Windows feature to support other remaining possible types such as time batch window, length batch window, time length window, unique window, first unique window, and external time window
- Join feature to support left outer, right outer, and full outer type queries
- Support ‘every’ keyword in Pattern feature query
- Time windows should be able to support expired event-processing scenario as discussed in Section 6.2.1

Implement extensive settings to cover most common use cases

As discussed in Section 6.2.2, the user requires some knowledge on Arduino programming to modify the Input/Output adaptor on the generated sketch in complex cases, as the current settings does not extensively cover most of the use cases in the Input/Output adaptor. As a result, the user requires modifying the generated Input/Output Adaptor, which necessitates the knowledge of Arduino programming.

Combine real time and historical data

To make the CEP to better it require meaningful data from everywhere so the data needs to correlated in several ways not only multiple streams and topics needs to be correlated similar to the current CEP engine. It also require Historical and real-time data needs to be correlated, and needs to correlate topics from other middleware and message-busses. For example, the temperature increase interval for the fire alarm depends on the historical temperature for that month, which needs to be coming from historical data.

Implement a mechanism to generate less readable and compiler friendly sketch source, which users cannot modify

Implementing extensive settings to cover most of the common use cases as discussed in Section 7.2.2, will drastically reduce the need for the user to modify the Input/Output adaptor. In addition, once the mechanism is implemented to separate the user modifiable sketch code from the actual final Arduino sketch as discussed in

Section 7.2.4, it will eliminate the need for the user to read the sketch and understand what was deployed in Arduino.

As a result, we can implement the mechanism to create the sketch by removing the formatting and whitespaces, system generated shorter non-meaningful variable, and method names. This code considerably reduces the final sketch size and increases the efficiency in compiling the sketch.

Optimizing power consumption

Majority of Arduino-based real-world applications may only run on batteries. Thus, the lower power consumption of Arduino program is important. One technique to reduce power consumption is to take out only the ATmega microcontroller from Arduino board and install on the breadboard (circuit board), once the program is uploaded to the chip. This helps in reduce power consumption since Arduino board consumes considerable amount of power to regulate the voltage, even during sleep time. Other technique that needs to part of the CEP engine for embedded device is to implement the ways to reduce power consumption of CEP Engine for embedded device and power consumption of the microcontroller. Few libraries helps in doing this, and one of the widely used library is JeeLib [43].

Support advanced queries

There are considerable number of real world use cases, which falls into the category of advanced queries. Advanced queries means the single query consist of sub-queries and/or having more than one feature in a single query. For example, there will be a use case that consists of ‘Filter’ to reduce the number of unnecessary events at the beginning, and then there will be a ‘Pattern’ type feature, which do the actual logic of the use case.

Implement proper error handling

The error inputs identified in the Siddhi Queries should be extensively handled to give the exact clear error message to the user with why, what, and how the reported error needs to be rectified. Proper error handling as such is one of the main requirements before it is applied in the production environment as an end product.

REFERENCES

[1] D. Evans, "The Internet of Things: How the Next Evolution of the Internet Is Changing Everything." Cisco Internet Business Solutions Group (IBSG), Apr-2011.

[2] P. Pietrzak, P. Lindgren, and H. Makitaavola, "Towards a lightweight CEP engine for embedded systems," in proc. *38th Annual Conference on IEEE Industrial Electronics Society (IECON 2012)*, 2012, pp. 5805–5810.

[3] X.-Y. Chen and Z.-G. Jin, "Research on Key Technology and Applications for Internet of Things," *Phys. Procedia*, vol. 33, pp. 561–566, 2012.

[4] A. de Saint-Exupery, *Internet of Things*. 2009.

[5] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Context aware computing for the internet of things: A survey," *Commun. Surv. Tutor. IEEE*, vol. 16, no. 1, pp. 414–454, 2014.

[6] C. C. Aggarwal, N. Ashish, and A. Sheth, *The Internet Of Things: A Survey From The Data-Centric Perspective*. Springer US, 2013.

[7] J. A. Stankovic, "Research Directions for the Internet of Things," *IEEE Internet Things J.*, vol. 1, no. 1, pp. 3–9, Feb. 2014.

[8] A. de Castro Alves, "New Event-Processing Design Patterns Using CEP," in proc. *Business Process Management Workshops*, 2010, pp. 359–368.

[9] S. Suhothayan, K. Gajasinghe, I. Loku Narangoda, S. Chaturanga, S. Perera, and V. Nanayakkara, "Siddhi: A second look at complex event processing architectures," in Proc. *ACM workshop on Gateway computing environments*, 2011, pp. 43–50.

[10] Sybase, "Analyze and Act on Fast Moving Data: An Introduction to Complex Event Processing." Sybase, 13-Jan-2012.

[11] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, W. M. White, and others, "Cayuga: A General Purpose Event Monitoring System.," in *CIDR*, 2007, vol. 1, pp. 412–422.

[12] D. Robins, "Complex event processing," in proc. 2nd *Intl. Workshop on Education Technology and Computer Science. Wuhan*, 2010.

[13] D. Gyllstrom, E. Wu, H.-J. Chae, Y. Diao, P. Stahlberg, and G. Anderson, "SASE: Complex event processing over streams," *ArXiv Prepr. Cs0612128*, 2006.

[14] S. Rizvi, "Complex event processing beyond active databases: Streams and uncertainties," Master's thesis, EECS Department, University of California, Berkeley, 2005.

[15] S. Wasserkrug, A. Gal, O. Etzion, and Y. Turchin, "Complex event processing over uncertain data," in *Proceedings of the second international conference on Distributed event-based systems*, 2008, pp. 253–264.

[16] "Complex Event Processor | WSO2 Inc." [Online]. Available: http://wso2.com/products/complex-event-processor/. [Accessed: 21-May-2015].

[17] O. M. de Carvalho, E. Roloff, and P. O. Navaux, "A Survey of the State-of-the-art in Event Processing," *11th Workshop Parallel Distrib. Process. WSPPD 2013*, p. 4, 2013.

[18] "EsperTech - Event Series Intelligence," *EsperTech - Event Series Intelligence*. [Online]. Available: http://www.espertech.com/products/esper.php. [Accessed: 20-Jun-2015].

[19] E. Alevizos and A. Artikis, "Being logical or going with the flow? A comparison of Complex Event Processing systems," in *Artificial Intelligence: Methods and Applications*, Springer, 2014, pp. 460–474.

[20] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik, "Scalable Distributed Stream Processing.," in *CIDR*, 2003, vol. 3, pp. 257–268.

[21] "The Aurora Project," *The Aurora Project*. [Online]. Available: http://cs.brown.edu/research/aurora/.

[22] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White, "Cayuga: a high-performance event processing engine," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007, pp. 1100–1102.

[23] J. Krämer and B. Seeger, "PIPES: a public infrastructure for processing and exploring streams," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, 2004, pp. 925–926.

[24] S. Grimm, T. Hubauer, T. Runkler, C. Pachajoa, F. Rempe, M. Seravalli, and P. Neumann, "A CEP Technology Stack for Situation Recognition on the Gumstix Embedded Controller." GI-Jahrestagung, volume 220 of LNI, page 1925-1930. GI, 2013.

[25] I. Zappia, F. Paganelli, and D. Parlanti, "A lightweight and extensible Complex Event Processing system for sense and respond applications," *Expert Syst. Appl.*, vol. 39, no. 12, pp. 10408–10419, Sep. 2012.

[26] "EsperTech - Products - Esper." [Online]. Available: http://www.espertech.com/products/esper.php. [Accessed: 11-Sep-2016].

[27] "Triceps." [Online]. Available: http://triceps.sourceforge.net/. [Accessed: 11-Sep-2016].

[28] L. Woods, J. Teubner, and G. Alonso, "Complex event detection at wire speed with FPGAs," *Proc. VLDB Endow.*, vol. 3, no. 1–2, pp. 660–669, 2010.

[29] D. K. Fisher and P. J. Gould, "Open-Source Hardware Is a Low-Cost Alternative for Scientific Instrumentation and Research," *Mod. Instrum.*, vol. 1, no. 2, pp. 8–20, 2012.

[30] J. M. Pearce, "Quantifying the Value of Open Source Hard-ware Development," *Mod. Econ.*, vol. 6, no. 1, pp. 1–11, 2015.

[31] A. Gibb, *Building open source hardware: DIY manufacturing for hackers and makers*. Pearson Education, 2014.

[32] "Arduino," *An Open-Source Electronics Prototyping Plat- form*. [Online]. Available: http://www.arduino.cc/. [Accessed: 17-May-2015].

[33] "First Steps With The Arduino: A Closer Look At The Circuit Board & The Structure Of A Program," *First Steps With The Arduino*. [Online]. Available: http://www.makeuseof.com/tag/steps-arduino-closer-circuit-board-structure-program/. [Accessed: 19-Jun-2015].

[34] "MSP430 LaunchPad," *MSP430 LaunchPad*. [Online]. Available: http://www.msp430launchpad.com/. [Accessed: 19-Jun-2015].

[35] "Wiring," *Wiring*. [Online]. Available: http://wiring.org.co/. [Accessed: 20-Jun-2015].

[36] "Pinguino," *Open Hardware Electronics Prototyping Platform Open Source Integrated Development Environment (IDE)*. [Online]. Available: http://www.pinguino.cc/. [Accessed: 20-Jun-2015].

[37] "Teensy," *USB-based microcontoller development system, in a very small footprint!* [Online]. Available: http://www.adafruit.com/products/199. [Accessed: 20-Jun-2015].

[38] "ANTLR," *ANTLR (ANother Tool for Language Recognition)*. [Online]. Available: http://www.antlr.org/. [Accessed: 17-May-2015].

[39] "Dynamic systems development method," *Wikipedia, the free encyclopedia*. 07-Jun-2015.

[40] "Google Java Style," *Google Java Style*, 30-Jun-2015. [Online]. Available: https://google-styleguide.googlecode.com/svn/trunk/javaguide.html.

[41] "Arduino Style Guide," *Arduino Style Guide*, 30-Jun-2015. [Online]. Available: https://www.arduino.cc/en/Reference/StyleGuide.

[42] "Build software better, together," *GitHub*. [Online]. Available: https://github.com. [Accessed: 18-Jun-2015].

[43] “jcw/jeelib,” *GitHub*. [Online]. Available: https://github.com/jcw/jeelib. [Accessed: 25-Jun-2015].