Complex Event Processing in ThingML

An Ngoc Lam(✉) and Øystein Haugen

Østfold University College, Halden, Norway
{anl,oystein.haugen}@hiof.no
http://www.hiof.no

Abstract. Complex Event Processing (CEP) is concerned with real-time detection of complex events within multiple streams of atomic occurrences. Numerous approaches in CEP have been already proposed in the literature. In this paper, we examine the CEP Extension of ThingML which is a cross-platform modeling language for deploying Cyber Physical System (CPS). In particular, we focus on both language characteristic and performance of the ThingML Extension while processing CEP queries. Experiments show that although ThingML does not outperform other well-known CEP engines, it is still a potential CEP solution for CPS which has limited physical resources. In addition, ThingML also shows its efficiency in term of language expressiveness in comparison with State Machine based CEP queries.

Keywords: ThingML Modeling Language · Complex Event Processing · Language expressiveness · Processing performance · Cyber Physical System

1 Introduction

Complex Event Processing (CEP) is a set of methods and techniques for tracking and analyzing real-time streams of information and detecting patterns or correlations of unrelated data (complex events) that are of interest to a particular business [21]. Besides being an attractive research topic, CEP concept is already applied in many areas such as finance, manufacturing processes, energy management, etc. [6]. It also has a strong impact on information systems design especially with the pervasive evolution of decentralized data nowadays [6]. Today, with the convergence of *Internet of Thing* (IoT) and *Big Data*, the ability to large-scale real-time stream processing and analysis become more and more demanding; especially in *Cyber Physical Systems* (CPS) where fast response is sometimes crucial (e.g., traffic management systems). Therefore, CEP has evolved to cope with these situations in order to build highly scalable and dynamic systems.

Although CEP systems have been designed to accomplish the same goal, they present different solutions regarding data model, processing algorithm and system architecture [8]. *Event Pattern Language* (EPL) which is the language to define atomic or complex event and specify the process of filtering (determine

J. Grabowski and S. Herbold (Eds.): SAM 2016, LNCS 9959, pp. 20–35, 2016.
DOI: 10.1007/978-3-319-46613-2_2

event of interest) and extracting events properties for constructing high-level events [19] is one of the key elements of the CEP solution. *RAPIDE-EPL* is an example of EPL with the ability to declare event types (integer, string, boolean, array, record) and attributes as well as matching rule [19]. *SASE* is a CEP system with a SQL-similar language that combines filtering, correlation, and transformation of RFID data for delivery of relevant, timely information as well as storing necessary data for future querying [25]. *The Cayuga System* is a high-performance system for complex event processing which has a well-defined query language for event pattern detection [9].

In recent years, *HEADS* project [1] has presented *ThingML* [2], a domain-specific language and compiler for the IoT, which includes concepts to describe both software components and communication protocols. ThingML provides developers the abilities to deploy the same implementation onto various platforms (Java, Javascript C/C++ and Arduino) as well as extend to new platforms. The formalism used is a combination of architecture models, state machines and an imperative action language. Recently, ThingML has been extended to include CEP capabilities supplementing the state machines. ThingML provides mechanisms to declare events, extracting attributes and some basic event operations such as: join, merge, filtering, etc.

This paper investigates CEP capabilities of ThingML. In particular, we aim to evaluate whether ThingML with CEP Extension could be sufficient for deploying CEP applications for CPS systems. Our approach is conducting a detailed study of both language characteristics and processing performance of this extension in order to answer following two research questions:

- **RQ$_1$.** Is ThingML CEP Extension an efficient language for developing CEP applications?
- **RQ$_2$.** Is ThingML CEP Extension powerful enough for CPS systems?

To answer RQ$_1$, we conduct a study analyzing how CEP operators would be described in ThingML using the CEP extension and comparing them with descriptions using only pure ThingML. Regarding RQ$_2$, a benchmark including data rate (events per second), latency (response time) and resource consumption is running for each CEP operator in order to evaluate the ability to execute complex event queries over real-time streams of sensing data.

2 Related Work

In the context of event processing systems, there are some frequently applied benchmarks that are relevant to CEP: the Linear Road benchmark for Stream Data Management Systems [4], the BEAST benchmark for Object-Oriented Active Database Systems [12,13], the SPECjms2007 benchmark for Message-Oriented Middleware [20], the NEXMark benchmark for Queries over Data Streams [23] and BiCEP - a CEP system benchmark [5]. However, currently there is no standardized benchmark that allows an objective comparison of different CEP systems.

In 2006, Wu et al. [25] presented SASE - a CEP system that executes monitoring queries over streams of RFID reading and provided a comparison between SASE and a relational stream processor TelegraphCQ [7] developed at the University of California, Berkeley. In this study, solely latency test was conducted to compare the performance of the two systems. The experiments showed that SASE performed much better than TelegraphCQ, eventually achieved much better scalability [25]. Later, Suhothayan et al. [22] when presenting Siddhi - a CEP Engine that incorporated several improvements including the use of pipeline architecture - also provided a comparison with Esper - the most widely used open source CEP Engine. Similar to previous work, the experiments conducted in this work only focused on latency metric as a criterion to evaluate the effectiveness of their proposed approach.

In 2007, Bizarro introduced BiCEP [5], a project to benchmark CEP systems. His main goal was to identify the core CEP requirements and to develop a set of synthetic benchmarks that allowed a comparison of CEP products and algorithms in spite of their architectural and semantic differences. In his paper, he described the design and the benchmark metrics such as: sustainable throughput, response time, scalability, adaptivity, computation sharing, etc. In the following years, Mendes et al. built FINCoS, a framework that provides a flexible and neutral approach for testing CEP systems [16]. FINCoS introduces particular adapters to achieve a neutral event representation for various CEP systems. In a further publication, Mendes et al. [18] used their framework to conduct different performance tests on three CEP engines - Esper and two developer versions of not further specified commercial products. In this work, they focused on the impact of variations of CEP rules by varying query parameters such as window size, windows expiration type, predicate selectivity, and data values. In a further work, Mendes et al. [17] introduced Pairs benchmark aiming at assessing the ability of CEP engines in processing progressively larger volumes of events and simultaneous queries while providing quick answers.

In the following years, there were also some works introducing benchmarks for CEP systems. However, similar to aforementioned works, they only concentrated on performance metrics as the criteria for evaluation. In 2011, Grabs et al. [14] proposed using metrics: data rate, latency and resource consumption to measure performance of CEP systems. In 2012, Wahl et al. [24] described their concept to measure the performance of different CEP systems in an automated manner by introducing a testing environment that included an event emitting component with stable interface, an interchangeable CEP component based on this interface and a measurement and evaluation component. Recently, in 2013, Mathew also conducted several experiments to evaluate the open source CEP system Esper based on four metrics: CPU utilization, Memory Utilization, Selectivity and Number of Classes [15].

In our work, we also perform experiments to evaluate the performance of ThingML on CEP capabilities. Although we measure metrics that aforementioned benchmarks also used [15,18], our work achieve a step further by more comprehensively focusing on language characteristic and application of

ThingML. In particular, we introduce several metrics to assess the expressiveness of the language and compare with ThingML without CEP features.

3 Background

3.1 CEP Operators

Complex Event Processing is one of the most rapidly emerging fields in data processing, and it is a principal technology solution for processing real time data streams [22]. A Complex Event Processor could be able to identify meaningful patterns, relationships and data abstractions from various streams of unrelated events. Once such information is extracted, the CEP engine would encapsulate it into a *composite event* and send to the interested components. To describe those behaviors, CEP uses a number of primitive operators as envisaged in [8]:

– *Selection* filters relevant events based on the values of their attributes. As an example, consider the following pseudocode pattern which selects `Thermometer` events that carry the temperature reading between 50 and 100.
  Pattern 1:
  ```
  Select Thermometer(temp >= 50 and temp <=100)
  From DataSource
  ```
– *Projection* extracts or transforms a subset of attributes of the events. For example, Pattern 2 selects only the humidity attributes of `Thermometer` events.
  Pattern 2:
  ```
  Select Thermometer(humid)
  From DataSource
  ```
– *Window* defines which portions of the input events to be considered for detecting a pattern.
– *Conjunction* considers the occurrences of two or more events. As an example, Pattern 3 can be used to capture a hypothetical event of `Fire` where both `Smoke` and high `temp` events are notified within the window frame of 5 min.
  Pattern 3:
  ```
  Within 5m. Smoke() and Thermometer(temp > 50)
  From DataSource
  ```
– *Disjunction* considers the occurrences of either one or more events in a predefined set.
– *Sequence* introduces ordering relations among events of a pattern which is satisfied when all the events have been detected in the specified order.
– *Repetition* considers a number of occurrences of a particular event. Pattern 4 illustrates an usage example of Repetition which detects a number of occurrences of high temperature.
  Pattern 4:
  ```
  Select Thermometer(temp > 60) as Temp
  From DataSource
  Where count(Temp) > 5
  ```

- *Aggregation* introduces constraints involving some aggregated attribute values. As an example, Pattern 5 computes the average value of humidity from `Thermometer` events.

  Pattern 5:
  ```
  Select avg(Thermometer.humid)
  From DataSource
  ```
- *Negation* prescribes the absence of certain events. Pattern 6 enhances the detection of `Fire` events by introducing the absence of `Rain` events.

  Pattern 6:
  ```
  Within 5m. Smoke() and Thermometer(temp > 50) and not Rain()
  From DataSource
  ```

A CEP query may contain several of these primitive operators in order to describe more complex patterns or behaviors. CEP engines provide the runtime to perform complex event processing where they accept these queries and match them against the event streams and trigger an event or execution whenever the conditions specified by the queries have been satisfied [22].

3.2 ThingML

ThingML is a domain-specific modeling language which provides a practical model-driven software engineering toolchain targeting resource constrained embedded systems such as low-power sensors, microcontroller based devices, gateways, etc. and facilitates their integration with more powerful resources (e.g. servers, cloud) [1,2]. ThingML provides mechanisms to describe both software components and communication protocols. The language also provides a template mechanism to integrate with third-party API, rather than re-developing them from scratch [1,2]. Currently, it supports transformation from ThingML model to targeting platforms such as C (Linux and Arduino) and Java.

ThingML language provides mechanism to describe the software components as state machine based models whose internal states and communication protocols are based on event triggers. Please refer to [2] for a full explanation of the language syntax and semantics, and [11] for the example of an adaptive temperature sensor network running on a microcontroller platform. Recently, in order to enhance the capability of event processing, ThingML has been extended with CEP logic. Currently, ThingML supports following operators:

- *Selection*: filters events according to their parameters, discarding elements that do not satisfy a given constraint. The following example presents a ThingML selection query which processes the stream of event `E1` from the event port `eventPort`, keeps only the events which have attributes values from 10 to 80, and forwards these events to `cep` port:

  ```
  stream SelectionStream
  from m : eventPort?E1::keep if (m.att1 > 10 and m.att5 < 80)
  produce cep!cepEvt(m.att1, m.att2, m.att3, m.att4, m.att5)
  ```

– *Projection*: extracts only part of the information contained in the event. As an example, it is used to extract and transform only attributes of interest:

```
stream ProjectionStream
from m : eventPort?E1
select var att1: Integer = m.att1 + 2
       var att2: Integer = m.att2 * m.att2
produce cep!cepEvt(att1, att2)
```

– *Conjunction*: A conjunction of events $E_1, E_2, ...E_n$ is satisfied when all the events $E_1, E_2, ...E_n$ have been detected (in any order). For example, the following code snippet illustrates the usage of conjunction to detect the occurrences of both events E_1 and E_2:

```
stream ConjunctionStream
from m : [e1 : eventPort?E1 & e2 : eventPort?E2
         -> cepEvt(e1.att1, e2.att1)]
produce cep!cepEvt(m.att1, m.att2)
```

– *Disjunction*: A disjunction of events $E_1, E_2, ...E_n$ is satisfied when at least one of the events $E_1, E_2, ...E_n$ has been detected. The following example illustrates the disjunction of E_1 and E_2:

```
stream DisjunctionStream
from m : [e1 : eventPort?E1 | e2 : eventPort?E2 -> cepEvt]
produce cep!cepEvt(m.att1, m.att2, m.att3, m.att4, m.att5)
```

– *Window*: defines which portions of the input flows have to be considered during the execution of operators. There are two types of windows supported by ThingML: time and length windows. The window attribute is defined by two values: size (time span) and step (time shift). As an example, the following code snippet illustrates the usage of window to compute the average, minimum and maximum values of the attribute `att1` of event E_1 within the window:

```
stream LengthWindowStream
from m : eventPort?E1 :: buffer 5 by 5
select var avg : Double  = average(m.att1[])
       var min : Integer = min(m.att1[])
       var max : Integer = max(m.att1[])
produce cep!cepEvt(avg, min, max)

stream TimeWindowStream
from m : eventPort?E1 :: during 5000 by 5000
select var avg : Double  = average(m.att1[])
       var min : Integer =  min(m.att1[])
       var max : Integer = max(m.att1[])
produce cep!cepEvt(avg, min, max)
```

4 A Study on CEP Functional Capacities of ThingML

This study aims to evaluate how CEP applications could be deployed with ThingML. Particularly, the study examines CEP capabilities of ThingML language by analyzing the capacities and performance of the ThingML CEP Extension. Detailed experiments are also conducted in order to compare ThingML CEP Engine with other existing CEP engines in term of language characteristic and processing performance.

In this study, we assess the CEP capability of ThingML based on the CEP operators that the language supports. In particular, we evaluate the using efficiency and processing performance of six CEP operators (as envisaged in [15]) which are currently supported by ThingML. The queries of all operators that are used throughout the experiments of this study are the ones presented in Sect. 3.2.

The following sections discuss the evaluation of language expressiveness and performance of ThingML CEP capacity in order to answer the two aforementioned research questions:

4.1 RQ_1: Is ThingML CEP Extension an efficient language for developing CEP applications?

By "efficient language", we mean an Event Pattern Language which provides concise and meaningful definitions of atomic and complex events. To answer this question, we present a language characteristic analysis of the CEP extension of ThingML. We demonstrate the language expressiveness of ThingML by comparing the CEP operators of ThingML with our implementation of the operators without using CEP extension in order to provide insights into the strength and limitations of the two implementation strategies. For evaluation, we use the following metrics for each of the specified queries:

– *Lines of Codes (LoC)*: number of lines of code written in both implementations (with and without using CEP extension).
– *Number of Keywords (NoK)*: number of keywords used in each implementation.

Table 1 shows the comparison of LoC and NoK between the two CEP queries implementations in ThingML. As can be seen from this table, the implementation of CEP queries with CEP extension always uses smaller number of code and keywords. The smallest and largest differences in LoC are 3 (Selection query) and 14 (Time Window query), the respective numbers in NoK are 6 (Selection query) and 45 (Conjunction query). Although these differences are not substantial, they do state the efficiency of using CEP capacity of ThingML in term of language expressiveness. In particular, by using CEP extension of ThingML to deploy CEP application, we could improve the conciseness of the source code, which could improve the understandability of the source code, produce less error or even save time for development.

Table 1. Language expressiveness measurement of ThingML with/without CEP.

Operator	LoC		NoK	
	With CEP	Without CEP	With CEP	Without CEP
Selection	55	58	54	60
Projection	61	64	50	58
Conjunction	62	98	50	95
Disjunction	60	64	58	70
Length window	104	126	110	135
Time window	105	119	115	136

```
internal event m:eventPort?E1 action do
   if(m.att1 > 10 and m.att5 < 80) do
      cep!cepEvt(m.att1, m.att2, m.att3, m.att4, m.att5)
   end
end
```

Fig. 1. Implementation of selection query without using CEP extension.

Table 1 only shows the differences of both implements of every single query, which leads to the small differences between the two columns. However, in practice, an application would contain more than one query or the expressions and computations of the queries could be much more complicated. Therefore, without CEP extension, these numbers could become considerably large.

Figure 1 shows the partial implementation of the selection query presented in Sect. 3.2 without using CEP extension. As can be seen, for this type of operator, there is not much difference between the two implementations even if the Boolean expression becomes more complex, resulting in the small difference between LoC and NoK.

However, for disjunction operator which is a similar type of occurrence detection, using CEP extension could be much more efficient. As can be seen from Fig. 2 which shows the partial implementation of disjunction query without using CEP extension, the occurrences of each event should be checked individually, which leads to code duplication and errors if the number of events in the disjunction query increases.

```
internal event m:eventPort?E1 action do
   cep!cepEvt(m.att1, m.att2, m.att3, m.att4, m.att5)
end
internal event m:eventPort?E2 action do
   cep!cepEvt(m.att1, m.att2, m.att3, m.att4, m.att5)
end
```

Fig. 2. Implementation of disjunction query without using CEP extension.

```
property att1 : Integer[1000]
property att2 : Integer[1000]
property i : Integer = 0
...
internal event e:eventPort?E1 action do
   att1[i] = e.att1
   att2[i] = e.att2
   i = i + 1
end
internal event timer?timer_timeout action do
   i = 0
   var avg : Double = average(att1)
   var min : Integer =  min(att1)
   var max : Integer = max(att1)
   cep!cepEvt(avg, min, max)
   timer!timer_start(1000)
end
```

Fig. 3. Implementation of time window query without using CEP extension.

The same problem of code duplication also occurs for time window and conjunction (Figs. 3 and 4). Especially for conjunction query which involves only two events, the implementation is relatively large. Thus, with the increase of the number of events, without using CEP Extension, the implementation of this operator would be much more cumbersome and difficult to manage. Moreover, for these types of operations, we should also use global variables for storing intermediate attributes. Thus, causing more memory consumption increasing with the number of involving events or the number of attributes of the events. Especially for time windows and length windows where the number of events in each window is undetermined, memory reservation for storing these events could be problematic as ThingML does not support dynamic allocation.

Currently, in the implementations of time window (see Fig. 3) and length window, we assume that the size and step of the windows are equal, thus, the windows do not overlap each other. For simplicity, we have not analyzed the other case as the algorithm could require more than two timers or become much more complicated. However, as presented above, this would be easily resolved by using CEP extension.

4.2 RQ_2: Is ThingML CEP Extension powerful enough for CPS systems?

To answer this question, we present a detailed performance analysis of ThingML. We also compare ThingML CEP extension with our implementation without CEP extension and Esper [10] which is an open source CEP Engine. We choose to compare to Esper because it is an open-source full-fledged stream processor. In addition, Esper is a Java-based software that has a well-supported user community, well-documented manuals, which facilitated this comparative study.

```
property isEvent1 : Boolean = false
property isEvent2 : Boolean = false
property event1Att : Integer
property event2Att : Integer
...
internal event e1 : eventPort?E1 action do
   isEvent1 = true
   event1Att = e1.att1

   if(isEvent2) do
      timer!timer_cancel()
      isEvent1 = false
      isEvent2 = false
      cep!cepEvt(event1Att, event2Att)
   end

   if(not isEvent2) do
      timer!timer_start(1000)
   end
end

internal event e2 : eventPort?E2 action do
   //analogous to waitE1 State
end

internal event timer?timer_timeout action do
   isEvent1 = false
   isEvent2 = false
end
```

Fig. 4. Implementation of conjunction query without using CEP extension.

Experiment Settings. All the experiments were performed on a workstation with a CPU Intel Core I5 2.60 GHz processor and 8 GB memory running Sun J2RE 1.8 on Window 10. We set the JVM maximum allocation pool to 1 GB, so that virtual memory activity has no influence on the results.

In order to test the system, we implemented an event generator that creates a stream of events with different throughputs from 1 to 1000 events per second. In our experiments, we considered 5 events types each of which has 5 attributes excluding the timestamps. For each attribute, the number of possible values (the domain size) was chosen from the range [1, 100]. We did not consider events with more attributes because the additional attributes were not used in our queries and can be projected out.

We measured the following metrics for each of the specified queries under different throughputs:

- *Latency* is the time taken to detect a complex event since the last event in the set of triggering events are sent to the CEP engine.

- *CPU Utilization* is the CPU Utilization for different kinds of CEP query over different event rates for a given pattern. It is measured by using a profiler called YourKit [3].
- *Memory Utilization* is the memory profile for different kinds of CEP query over different event rates for a given pattern; which is also measured with YourKit.

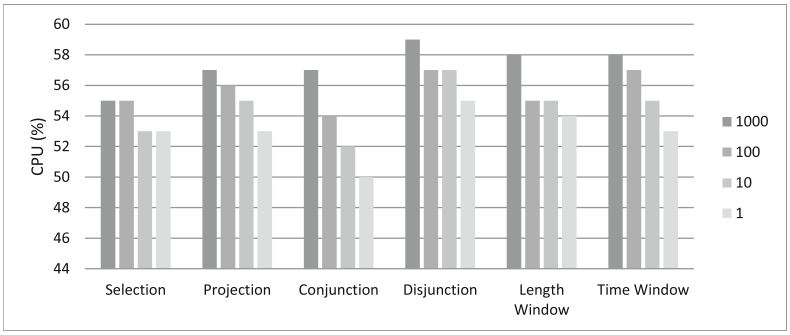
The criterion for stopping each experiment was such that the system has detected 1000 instances of the complex event specified in each query. The latency was obtained by averaging the latency values of 1000 runs. For memory utilization we captured the maximum heap memory allocated during the runs.

Result. Figure 5 presents the performance analysis of ThingML CEP Extension of the specified queries. As can be seen, the CPU utilization is around 50–60% which is medium CPU requirement. The CPU performance increases very slightly as the throughput changes from 1 to 1000, which shows that CPU requirement is not substantially affected by the occurrences of the events. For all types of queries, the CPU utilization is always at highest performance when the throughput is at highest rate (1000 events/s). This could be easily explained as the more processing performance is required when the events occur more frequently.

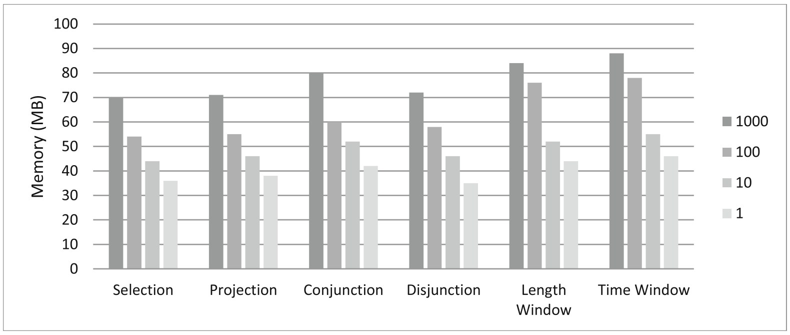
Similarly, the memory requirement for CEP Extension is also relatively low (around 35–90 MB). Memory usage is also increasing as the throughput rate raises. Length window and time window queries always amount to the largest memory because these queries need to store information of the events occur within the windows. In contrast, as the throughput increases, the latency is found to decrease. Which means as the events occur more frequently, the event processing time becomes shorter. This shows the effectiveness of the CEP engines which try not to lose too much meaningful information as the events come out so close to each other.

For the implementation without CEP Extension and Esper, the same pattern also happens. Therefore, for saving space, we do not include all the bar charts into this paper. Instead, we only show the comparison of the three implementations under the highest throughput (see Fig. 6). However, from the ThingML implementation without CEP Extension, we observe that there are remarkable differences in the three metrics. In particular, this implementation always requires slightly more physical resources (CPU, Memory) while the latency is slower. The differences could be clearly observed in conjunction, time window and length window queries which require more processing performance. This finding is also consistent with our discussion from the language characteristic analysis that our own implementation without using CEP extension may not be effectively optimized for CEP operations.

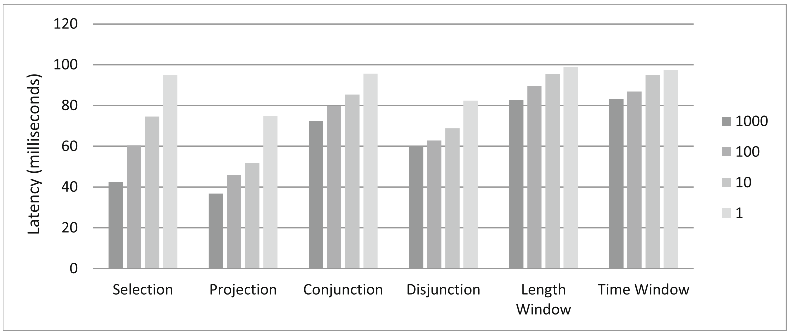
Compared with the other two implementations, Esper engine requires much more physical resources: CPU utilization is ranging from 75–90 % and memory usage is around 50–120 MB. However, the latency is much better (approximately



(a) CPU Usage (%)



(b) Memory Usage (MB)



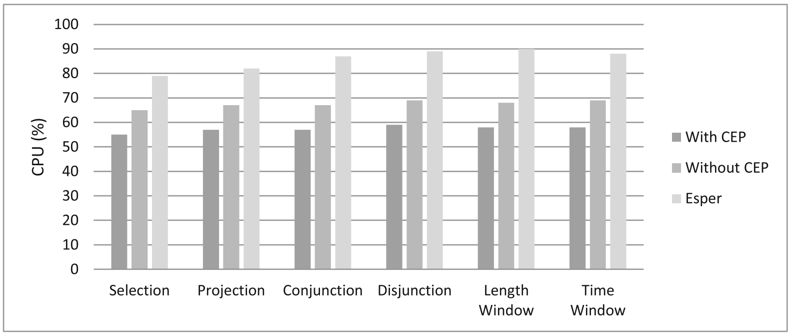
(c) Latency (millisecond).

Fig. 5. Performance analysis of ThingML with CEP Extension.

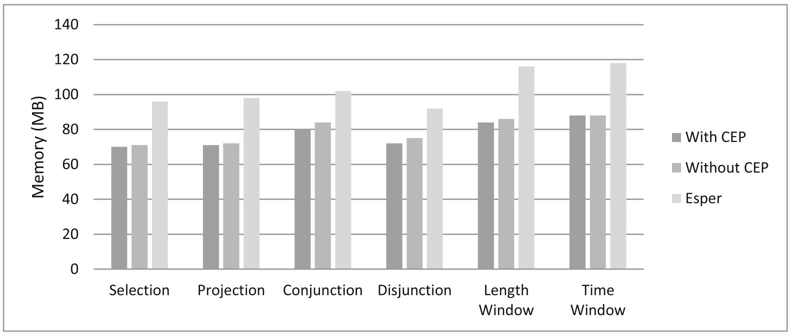
two times faster than CEP extension), which shows the effectiveness of the architecture (event processing algorithm, data structure, etc.) of this CEP engine.

4.3 Discussion

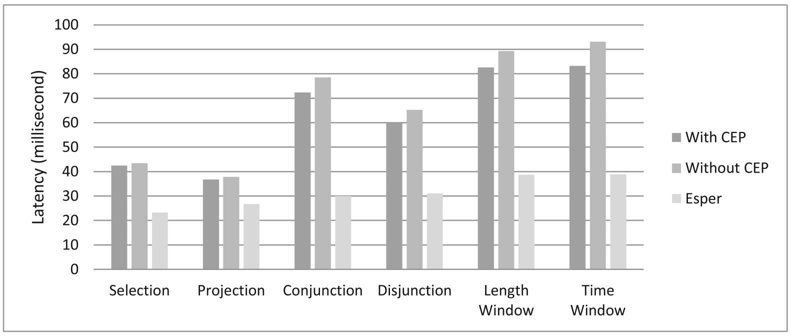
In this study, we perform the analysis of language characteristic and processing performance of ThingML CEP extension. Finding from the first analysis reveals



(a) CPU Usage (%)



(b) Memory Usage (MB)



(c) Latency (millisecond).

Fig. 6. Performance analysis of different implementations (1000 events/sec).

that the source code written by using CEP extension is much shorter and more concise than the implementation without using the extension, hence improving the understandability, saving development time, removing code duplication and less error prone. In this study, we only evaluate every single query separately and the queries are also simple, thus the differences between the two implementations may not be substantial. However, a real CEP application could involve tens of queries. Therefore, in these practical situations, using CEP extension would improve the efficiency of the development process. Moreover, CEP applications could also involve hundreds of event types with really complicated patterns for complex events (complex boolean expressions, nested operations, etc.) which may not be implemented without using CEP extension. Therefore, this CEP extension not only helps to save much more effort but also enables the implementation of CEP applications.

As can be seen from the second experiment, CEP extension of ThingML is not the most effective CEP Engine. However, this implementation requires relatively low physical resources which are extremely limited for CPS devices. In addition, although the latency of ThingML CEP is larger than that of Esper, it is still within the range required for CPS applications, which need responses within milliseconds. Also, because the event generator and CEP processor were implemented in the same application, the measured performance and physical consumption for CEP Engine also contained those of the event generator. Thus, the actual numbers for the CEP engine could be even smaller than those presented, which could be a threat to validity. However, because all tested CEP applications contain the same implementation of the event generator, the existence of the event generator does not influence the value of this comparative study, but rather emphasizes the power of the extension to perform CEP tasks.

5 Conclusion

In this paper, we presented the analysis of ThingML CEP extension, a complex event processing capacity of the modeling language for embedded and distributed systems. We first assessed the language expressiveness of this CEP extension by considering the two quantitative attributes of the source code written with and without the CEP extension. The assessment revealed that by using CEP extension, CEP application written in ThingML could be much more concise. In addition, this capacity could enable the implementation of some complex event patterns which require complicated algorithm or even could not be implemented without using CEP extension. We also performed the analysis of physical resource consumption and processing performance of ThingML in comparison with usual implementation (without using CEP extension) and Esper. Findings from this experiment also showed that ThingML required much smaller physical resources and reasonable latency values, which makes it a potential language for deploying CEP applications for CPS systems.

For future study, we should also need to evaluate the performance of a real ThingML CEP application which involves a variety of complex events because currently our experiments were tested on only single and simple queries. In addition, as presented above, ThingML is a modeling language which could be deployed on different physical platforms. However, in this paper, we only tested the performance experiments on computer workstation which has generous physical resources. Therefore, it is also necessary to consider the performance of this language on different CPS platforms (e.g., Arduino, Raspberry Pi).

References

1. Head project. http://heads-project.eu/
2. ThingML. http://thingml.org/pmwiki.php
3. Yourkit profiler. https://www.yourkit.com/
4. Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A.S., Ryvkina, E., Stonebraker, M., Tibbetts, R.: Linear road: a stream data management benchmark. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases, Vol. 30, pp. 480–491. VLDB Endowment (2004)
5. Bizarro, P.: BiCEP-benchmarking complex event processing systems. In: Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2007)
6. Buchmann, A., Koldehofe, B.: Complex event processing. IT-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik **51**(5), 241–242 (2009)
7. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S.R., Reiss, F., Shah, M.A.: TelegraphCQ: continuous dataflow processing. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, p. 668. ACM (2003)
8. Cugola, G., Margara, A.: Processing flows of information: from data stream to complex event processing. ACM Comput. Surv. (CSUR) **44**(3), 15 (2012)
9. Demers, A.J., Gehrke, J., Panda, B., Riedewald, M., Sharma, V., et al.: Cayuga: a general purpose event monitoring system
10. EsperTech: Esper (2016). http://www.espertech.com/esper/. Accessed 3 Feb 2016
11. Fleurey, F., Morin, B., Solberg, A.: A model-driven approach to develop adaptive firmwares. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011, pp. 168–177. ACM, New York (2011). http://doi.acm.org/10.1145/1988008.1988031
12. Geppert, A., Berndtsson, M., Lieuwen, D., Zimmermann, J.: Performance evaluation of active database management systems using the beast benchmark. Technical report. Citeseer (1996)
13. Geppert, A., Berndtsson, M., Lieuwen, D.F., Roncancio, C.: Performance evaluation of object-oriented active database systems using the beast benchmark. TAPOS **4**(3), 135–149 (1998)
14. Grabs, T., Lu, M.: Measuring performance of complex event processing systems. In: Nambiar, R., Poess, M. (eds.) TPCTC 2011. LNCS, vol. 7144, pp. 83–96. Springer, Heidelberg (2012). doi:10.1007/978-3-642-32627-1_6
15. Mathew, A.: Benchmarking of complex event processing engine-esper. Technical report, Technical Report IITB/CSE/2014/April/61, Department of Computer Science and Engineering, Indian Institute of Technology Bombay, Maharashtra, India (2014)
16. Mendes, M., Bizarro, P., Marques, P.: A framework for performance evaluation of complex event processing systems. In: Proceedings of the Second International Conference on Distributed Event-Based Systems, pp. 313–316. ACM (2008)
17. Mendes, M., Bizarro, P., Marques, P.: Towards a standard event processing benchmark. In: Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering, pp. 307–310. ACM (2013)
18. Mendes, M.R.N., Bizarro, P., Marques, P.: A performance study of event processing systems. In: Nambiar, R., Poess, M. (eds.) TPCTC 2009. LNCS, vol. 5895, pp. 221–236. Springer, Heidelberg (2009). doi:10.1007/978-3-642-10424-4_16

19. Robins, D.: Complex event processing. In: Second International Workshop on Education Technology and Computer Science. Wuhan (2010)
20. Sachs, K., Kounev, S., Bacon, J., Buchmann, A.: Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark. Perform. Eval. **66**(8), 410–434 (2009)
21. Schmerken, I.: Deciphering the myths around complex event processing. Wall Street & Technology, May 2008. http://www.wallstreetandtech.com/latency/deciphering-the-myths-around-complex-event-processing/d/d-id/1259489?
22. Suhothayan, S., Gajasinghe, K., Loku Narangoda, I., Chaturanga, S., Perera, S., Nanayakkara, V.: Siddhi: a second look at complex event processing architectures. In: Proceedings of the 2011 ACM Workshop on Gateway Computing Environments, GCE 2011, pp. 43–50. ACM, New York (2011). http://doi.acm.org/10.1145/2110486.2110493
23. Tucker, P., Tufte, K., Papadimos, V., Maier, D.: Nexmark-a benchmark for queries over data streams (draft). Technical report, OGI School of Science & Engineering at OHSU, September 2008
24. Wahl, A., Hollunder, B.: Performance measurement for cep systems. In: Proceedings of the 4th International Conferences on Advanced Service Computing, pp. 116–121 (2012)
25. Wu, E., Diao, Y., Rizvi, S.: High-performance complex event processing over streams. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 407–418. ACM (2006)