DEBS Grand Challenge: Event Processing of Monitoring Data of Large Hi-Tech Manufacturing Equipment

Kangheng Wu[1], Xiaokang Xiong[1], Bert W. Leung[1], Jihyoun Park[2], Zhibin Lei[1]
[1]Hong Kong Applied Science and Technology Research Institute
3/F, Bio-informatics Centre, 2 Science Park West Avenue
Hong Kong Science Park, Shatin, Hong Kong
[2]Chinese University of Hong Kong
Department of Information Engineering, Room 834, Ho Sin Hang Engineering Building
The Chinese University of Hong Kong, Shatin, N.T., Hong Kong
{khwu, xkxiong, bertleung}@astri.org, april3@gmail.com, lei@astri.org

ABSTRACT
The subject of DEBS 2012 Grand Challenge is to implement an automatic event-monitoring system that traces the state of Chem Additive sensors and their associated valves and records the energy consumption of the manufacturing equipment. As solutions for the problem, we compare three different approaches. The first one is a custom solution using optimal technologies, such as lock-free queue and minimum heap-list. The second one is a rule-based solution using JBoss Drools Expert. The third one is a CEP-enhanced rule-based solution using JBoss Drools Fusion. After evaluating the event processing performance among these three implementation methods in terms of throughput and latency, we finally conclude this paper with the discussion of implications of our solution methods including non-performance factors.

Categories and Subject Descriptors
H.2.4 [**Database Management**]: Systems – *rule-based databases, query processing.*

General Terms
Algorithms, Performance, Experimentation.

Keywords
Stream database, Rule engine, Complex Event Processing

1. INTRODUCTION
The quest of this year's DEBS Grand Challenge is the event stream processing on the monitoring data of large high-tech manufacturing equipment. The number of monitoring data recorded by the equipment is as many as 100 records per second. Each data as an event is transmitted via TCP/IP to an event processing system to be interpreted and trigger appropriate actions. By doing so, the Challenge addresses issues of real-time event stream processing with a huge input stream [4].

DEBS'12, July 16–20, 2012, Berlin, Germany.

The streaming database research is a relatively new research field that supports continuous data queries on stream data input. As the speed and size of the stream data input to the database system far exceed the capacity of the conventional database system to manage, new techniques that handle streaming data and enable on-demand query execution have come into request. Recently, its research arena has been broadened to the distributed streaming database. Moreover, various tracking statistics techniques were developed to enable more advanced management of the streaming model [3].

Our team took three different approaches to solve the problem. First, we implemented a custom solution based on our direct understanding about the operator description. We designed the algorithm to use optimal technologies such as the lock-free queue and the minimum heap-list for the event stream handling. Second, we used a rule engine, JBoss Drools Expert, since rule engines are known specialized and fast to execute conditional actions such as if/else statements on a large volume of input data [8]. Third, we used a CEP engine of JBoss Drools: Fusion. JBoss Drools Fusion provides easy development and maintenance tools to implement event-based systems, since it equips with sophisticated event processing functions. However, as the embedded event processing functions are not optimized well and require heavy supporting engines, it may suffer from considerable system overhead and latency issues. This is the reason why we tested the JBoss Drools-based solutions with and without the embedded event processing functions and compared the performances of two cases.

The remainder of this paper is organized as follows: Section 2 clarifies the definition of DEBS 2012 Grand Challenge's problem. Section 3 explains the principle and advantages of rule engines as event processing systems and introduces JBoss Drools that was selected in this paper to implement the solution. The implementation details of both custom and rule-based methods are described in Section 4. Section 5 displays the performance evaluation results of each implementation in terms of throughput and latency. Finally in Section 6, the paper concludes with a discussion of the implications of our findings including non-performance factors.

2. PROBLEM DEFINITION
The DEBS 2012 Grand Challenge requests to design a system that keeps tracing the monitoring data of a number of sensors of a manufacturing equipment, which are crucial indicators for ensuring the normal operation of the equipment. The monitoring

data are given in the form of a CVS file. By using the given generator, each line of the file, as an independent event, is sent to a network server at short intervals. Our system is required to receive these events and calculate the trends of the values in real time to take appropriate actions (Figure 1). Among various monitoring data contained in the original data packet, we use the time stamp, three sensor readings of power consumptions, three sensor readings of chemical additive and those of the associated valves controlled by the chemicals.

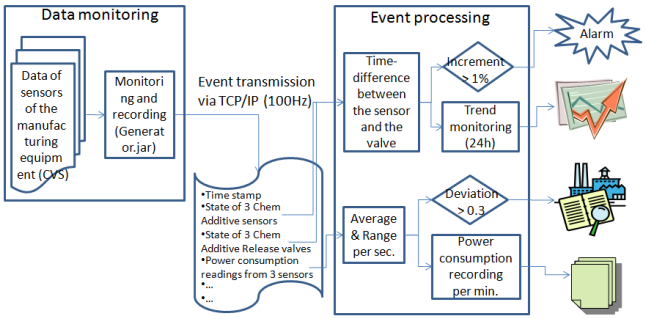

Figure 1. Event monitoring and processing flowchart

The problem set of DEBS 2012 is consisted of two queries: First, our system needs to observe the time difference between the state change of the Chem Additive Release valves and their control chemicals. While tracking the 24-hour trends of the value using the linear least square method, if an abnormality is detected, our system raises an alarm. An abnormal behavior is defined as the case that the difference between any two time difference values in the given time window exceeds 1%. To clarify the ambiguity, we assumed that we raise an alarm if any recorded time difference exceeds 1% of the minimum value in the last 24 hours. Second, our system reads the energy consumption levels of the manufacturing equipment, which are captured from 3 sensors. At every second, the average and relative variation are calculated for the data received during the time. If the deviation exceeds 30%, the raw data from 20 seconds before the moment to 70 seconds after it are forcibly recorded for further investigation. In addition, the amount of power consumption is printed per minute.

3. RULE-BASED SYSTEMS

This section explains the principle of rule engines and the selected features of JBoss Drools in regard with event stream processing.

3.1 The Principle

A rule-based approach provides a way to simply define complex problems using declarative statements. This is especially good when there need too many if/else conditions to solve a problem. Besides, due to its ability to separate dynamic knowledge base (data) and business logic (rules) from static application code, it allows easy and fast development in handling a large volume of data input. The basic architecture of rule engines for stream processing is depicted in Figure 2. Scrutinizing the input streaming data, if the input matches rules stored in the rule base, the rule engine fires the rules in the predefined order to execute certain actions or draw inference.

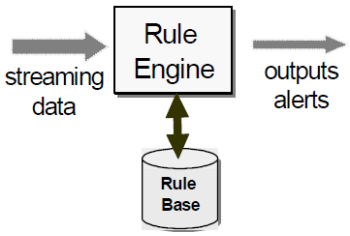

Figure 2. Basic architecture of a rule engine for stream processing [9].

3.2 JBoss Drools

JBoss Drools is a production system-type of rule engine using forward chaining. The inference logic of JBoss Drools starts from the known facts (data). If facts are changed by insertion, modification or retraction, the inference engine immediately searches rules corresponded with the given data using the pattern matching algorithm adopted from the Rete algorithm [5]. The Rete algorithm internally manages a network of nodes that have records of facts and their corresponding patterns for fast and efficient pattern matching. Since it sacrifices memory over speed, the algorithm may cause memory issues in a large system [1]. The basic package of rule engine in JBoss Drools is *Expert*. Regarding the complex event processing, it provides another specialized package: *Fusion*. Fusion extends the basic module by introducing the concept of event as a special case of fact [6]. Especially, it supports straightforward implementation on event stream processing by providing ready-made components of temporal reasoning based on [2, 10].

The structure of a rule has two sections: *when* and *then* (Figure 3). The *when* clause defines conditions of this rule to be fired in the declarative form. The *then* clause contains java codes commands to be executed. Since JBoss Drools is a Java-based open source rule engine, it provides full flexibility for developers to add custom Java codes and can be integrated in any java application.

```
rule "rule-name"
when
      <query in an Object query language>
then
      <any java code>
End
```

Figure 3. Basic template of a rule.

4. IMPLEMENTATION

This section describes the implementation details of our solutions for the event-monitoring system, focusing on the difference in how to handle event stream in each solution.

4.1 Custom Method (Optimized)

4.1.1 Overview

As a reference point for our rule-based methods, a custom implementation is built using C++. C++ is chosen because it is a highly customizable programming language for general purpose

and it is efficient to execute. The implementation closely followed the description of both queries with the help of lock-free queue and a data structure designed by the authors to provide satisfying throughput and latency, with an expense of memory usage.

4.1.2 Implementation

The implementation basically divides into three parts: a receiver, a query one handler and a query two handler. Each runs in separate threads. The receiver accepts data from network, decodes it following Google protocol buffer specifications, and passes the decoded data to query handlers. Both query handlers process the input data as defined by the requirements and provide output. The major challenges lie in how to pass and process data efficiently. They will be discussed separately.

As the data is rushing in with a speed of 100Hz or even faster, the receiver must process each data chunk as fast as possible with minimum delay. In the implementation, each query handler shares a channel with the receiver so that the receiver can continue pushing data in and at the same time the query handler gets and processes the data. Normally this is achieved by using a thread-safe queue and this kind of queue achieves thread-safe by using Mutex locks. While Mutex locks can help achieve concurrency, however, they are also one of the major performance blockades of the implementation because acquiring and releasing a Mutex lock is actually an expensive action in terms of number of CPU instruction involved.

As a result, a lock-free queue between the receiver and each query handler is necessary to achieve better performance. Luckily, the above can be viewed as requiring a lock-free queue with a single producer and a single consumer, which is a special case of lock-free compared with general case that involves multiple producers and multiple consumers. With the use of lock-free queue, the processing speed of the receiver can reach over 9000 tuples per second, compared to around 6000 tuples per second if Mutex lock is used.

Another challenge lies in processing data in query one, especially for evaluating operators 10, 12 and 14. According to the requirement, the data within the past 24 hours has to be checked and see if there is an increase of 1%. This requirement can actually be simplified as finding the minimum of the past 24 hours and if the latest value increases more than or equal to 1% of the minimum value, the alarm is activated. Assumes the data rushes in with a rate of 100Hz, there will be as many as 8640000 values within a 24-hour interval. This can be a problem if one fails to choose a proper data structure.

Speaking of finding the minimum value in a changing data set, one may easily think of using a minimum heap. However, it may not be trivial to update the heap with a sliding window of 24 hours. On the other hand, one of the most appropriate data structures to implement the sliding window is queue. However, the complexity of finding minimum value of a queue is O(n) which will be too slow for this problem. To solve this problem, minimum heap and queue are combined into one single data structure that provides operations of both of them. The idea is to arrange all nodes of a minimum heap into a linked list. Using this combined "minimum heap-list", we can find the minimum value in O(1) and both enqueue and dequeue can be achieved in O(log n).

4.2 Rule-based Method

4.2.1 Overview

As introduced in Sec. 3.2, a JBoss Drools rule engine contains a working memory that stores all the facts and a rule base engine that controls the program logic. With the help of working memory, we save lots of effort to design complex data structures to store the incoming data. We insert the data as facts into the working memory when data comes in, and retract it whenever it is no longer required in the quires. On the other hand, the program logics have to be implemented in rule language. Drools rules provide many powerful built-in functions to process the data (like max, min and average in the accumulate function), which also saves lots of implementation effort. For some operations not provided in the existing drools engine, we wrote our implementation as an extension to drools engine.

4.2.2 Expert Implementation

The data structure of the received message contains many fields, and most of the fields are not related to either Query 1 or Query 2. In this implementation, we strip the received message, use java objects to store the striped message and then insert the objects as facts to the working memory, in order to reduce the storage overhead.

All the processing on the data or states have to be done through rules, since the data and states stored as facts in the working memory are not accessible from the outside programs, For this reason, when any operator in either Query1 and Query 2 triggers, unlike conventional programming, which make functions calls, we need to insert a fact into the working memory to generate an event, and write a rule to handle this event.

Specifically, for both Query 1 and Query 2, we need to monitor the data over a period. To calculate the time period and raise a trigger when times up, we use one Java Class “Timer” to store start time and end time of each period, and another Java Class “TimeUp” to represent a time up event. When first message comes, we insert a “Timer” object into the working memory, and whenever a message comes with timestamp exceeds the end time of the time period, insert a “TimeUp” object which would consequently trigger out a rule to handle the time up event. The rule to implement this timer is displayed as follows.

```
rule "start every second"
  salience 2
  when
    $s : Timer( $endTs : endTs )
$m : Query2Message( ts > $endTs,
   $ts : ts)
  then
    modify( $s ){ setStartTs($ts) };
    if ($endTs > 0) {
      insert(new TimeUp($endTs));
    }
End
```

In rules, many of the operations on data can be completed by calling Drools native functions. To calculate the max, min, and average of the sets of monitoring data, we take heavy use of *accumulator* in Drools. With these built-in functions, the engineering efforts reduce a lot. However, we also notice the accuracy of the results is not guaranteed when using *accumulator* for long types. The last two digits of the long type looks to be random.

In addition, to complete custom operations on data/facts in the working memory, Drools provides an interface in Java for *accumulate* functions, which turns extension of the rule engine to an easy job. To plot the trends of delta in the 24 hours as required in operators 11, 13, and 14 in Query 1, we write an Accumulate Function for that purpose. Therefore, even though engineers do not have direct access to the facts in the working memory, they can easily extend the drools engine for their requirements.

4.3 CEP-enhanced rule-based Method

4.3.1 Overview

An event is considered as a special type of fact in JBoss Drools Fusion. According to [7], these events are usually immutable, have strong temporal constraints and managed lifecycle, and use sliding windows. Our last implementation for the Challenge problem uses these event processing features of JBoss Drools Fusion to handle the stream data.

4.3.2 Fusion Implementation

- **Events**

We treat the messages as a Java class and declare them as events in the rule file. The following is an example for Query1. Qurey1Event2_1 is a Java class and is declared as an event in the rule file. Its role is "event". It will be automatically expired after 24 hours from the time point when inserted into the working memory.

```
declare Query1Event2_1
  @role( event )
  @expires ( 24h )
end
```

- **Session Clock**

Reasoning over time requires a reference clock. From the QA section, we know the session clock of the CEP engine is the timestamp of the sensor reading that caused the violation. We use pseudo clock as follows.

```
KnowledgeSessionConfiguration sessConfig =
  KnowledgeBaseFactory.newKnowledgeSessionConfiguration();
sessConfig.setOption(ClockTypeOption.get("pseudo"));
... ...
clock.advanceTime(queryMsg.s_ts - tsPrevious,
  TimeUnit.NANOSECONDS);
```

- **Stream Support**

Most CEP use cases have to deal with streams of events. Drools Fusion generalizes the concept of a stream as an "entry point" into the engine. Instead of inserting events directly into the working memory, we insert events into the entry point as follows.

```
entryPointQuery1 =
session.getWorkingMemoryEntryPoint(ENTRY_POINT_QUERY1);
```

- **Temporal Reasoning**

Temporal reasoning is a requirement of any CEP engine. It can be applied in the system in which the relationship of events is strong temporal. We describe the first operator in Query 1 as follows.

```
rule "query 1 operator 1"
  salience 3
  no-loop true
    when
      $query1Event0_0 : Query1Event0_1(
        $s_bm_0 : s_bm,
        $s_ts_0 : s_ts,
        $index_0 : index
      ) from entry-point "query1_input"

      $query1Event0_1 : Query1Event0_1(
        this after [1ms, 20ms] $query1Event0_0,
        $s_bm_1 : s_bm,
        $s_ts_1 : s_ts,
        $index_1 : index
      ) from entry-point "query1_input"

      $query1Fact1 : Query1Fact1_1()

    then
      if (($s_bm_0 == false && $s_bm_1 == true)
        || ($s_bm_0 == true && $s_bm_1 == false)){
        printOperator1_2_3_4_5_6(1, $s_bm_1, $s_ts_1, $index_1);
        modify($query1Fact1){s_edge = $s_bm_1, s_ts = $s_ts_1,
          index = $index_1, state = 1};
      }

      retract ($query1Event0_0);
end
```

- **Event Processing Modes**

There are two main requirements to use stream mode. First, events in each stream must be time-ordered. Second, the engine forces synchronization between streams through the use of the session clock. For solving the Query1 and Query2, we enable the stream mode in the application as follows.

```
KnowledgeBaseConfiguration config =
KnowledgeBaseFactory.newKnowledgeBaseConfiguration();
config.setOption( EventProcessingOption.STREAM );
```

- **Sliding Windows**

Sliding windows is a way to scope the events of interest as the ones belonging to a window that is constantly moving. The two most common sliding window implementations are time based windows and length based windows. We describe the tenth operator in Query 1 as follows.

```
rule "query 1 operator 10"
  salience 1
  when
    $query1Operator10 : Query1Operator10(
      $s_dt_cur : s_dt,
      $s_ts_cur : s_ts,
      $index_cur : index
      )
      $query1Event2 : Query1Event2_1() from
      entry-point "query1_operator10"
    $s_dt_min : Number()
      from accumulate(
        Query1Event2_1(
          $s_dt : s_dt
        )
      over window:time(24h) from
        entry-point "query1_operator10",
      min($s_dt)
    )

  then
    retract ($query1Operator10);
    if (($s_dt_cur - $s_dt_min.longValue()) >
(Math.abs($s_dt_min.longValue()) * 0.01)){
      printOperator10_12_14(10, 1, $s_ts_cur);
      printStreamQuery1Operator10.printf("%d %d\r\n", 1, $s_ts_cur);
      printStreamQuery1Operator10.flush();
    }
end
```

- **Memory Management for Events**

From the manual in JBoss Drools Fusion, the engine can retract the event automatically when an event can no longer match any rule due to its temporal constraints.

- **Issue in Rule Triggered by Timer**

There is a problem in the rule which is triggered by timer. We find that the problem is using timer in rule in the entry point. When a rule is triggered by timer and adds a new event into the entry point, the rule will be triggered again and again until the pseudo clock is updated. For solving these problems, we do not use timer in the rule which adds a new event into the entry point, but we added another rule to trigger this event.

5. RESULTS AND EVALUATION

In this section, we assess the correctness of our solutions with respect to the problem specification and evaluate their performance in terms of throughput and latency.

We ran all of our experiments on a laptop computer (Lenovo T420) with a 4GB memory and Intel CORE i5 2.5GHz CPU using the Windows7 Operation System.

5.1 Correctness

According to our understanding from the problem description, we can get the same results from the custom, rule-based and CEP-enhanced solutions, that is, the correctness rate is 100%.

5.2 Throughput

The throughput parameter is defined as the number of handled input messages per second. We run each program to handle 1 million input messages. We have chosen 15 intervals of the speedup factor up to 200 in order to evaluate our solutions.

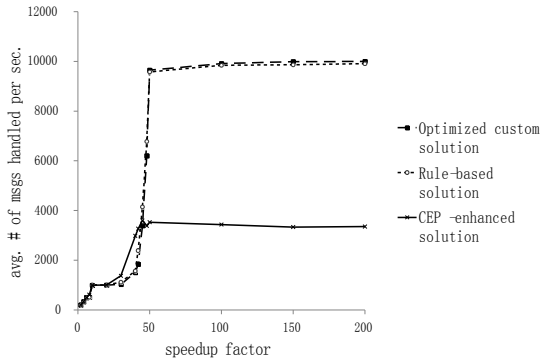

Figure 4. Throughput.

In our experiment, the data generator reaches its max output speed when the speedup factor is set as 50. As shown in Figure 4, our C++ and basic rule-based solutions can handle the maximum speed of the generator. The constant throughput after the speedup factor 50 is due to the capacity limitation of the generator, therefore we cannot test the maximum throughput of those solutions. However, the CEP solution is found to be able to handle the incoming data-speed up to the factor of 45 only. Although the performance of all three solutions are comparable before the point, the CEP solution cannot respond to higher incoming speeds. The implementation difference between the basic rule-based solution (JBoss Drools Expert) and the CEP-enhanced solution (JBoss Drools Expert + Fusion) lies on the timer function. The CEP engine uses the system or pseudo time to define the lifetime and duration of events and retracts events when the time is up. In the rule-based solution, we implemented our own timer to control the lifetime of facts in the knowledgebase. We use the logical time, which is based on the timestamp of the incoming data. The implementation is straightforward. It would respond whenever a new data comes in. It could not be probed fully due to the time limit, but is highly suspected that the performance bottleneck of the CEP solution comes from its own limitation in the timer engine to handle the lifetime of events.

5.3 Latency

The latency parameter is defined as the difference between the timestamp of receiving a message and the timestamp of finishing handling the message. We calculate the average time of handling 1 million messages measured in seconds for the latency parameter.

As the result of our evaluation on latency shows in Figure 5, the optimized custom solution produced the best performance. Its average latency for all the test conditions is 0.9 millisecond, thus it does not cause any severe latency issue. We can also observe that the performance of the CEP-enhanced rule-based solution is worse than that of the simple rule-based solution.

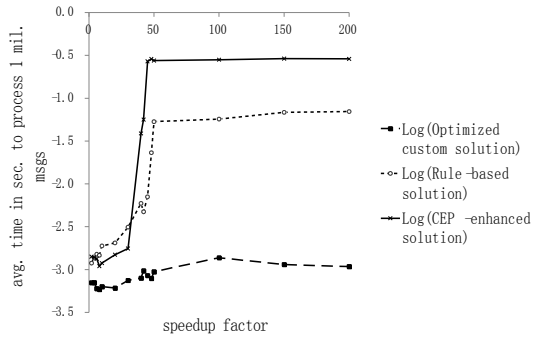

Figure 5. Latency.

For the test, we use a buffer to store the incoming messages in order that the unhandled messages do not block TCP/IP and in consequence drop down the generator's speed. While the generator keeps pushing data into the buffer, the buffer passes the incoming messages to the solution programs one by one. Our C++ solution uses an adapted and advanced buffer, lock-free queue, which was explained in section 4.1. The latency of the CEP solution increases drastically from the speedup factor of 30. The constant outcome of CEP solution after the speedup factor 45 is due to the limit of the buffer size. We imposed limitations on the buffer size because the latency would increase infinitely without it if the CEP solution reaches the processing capacity so that the number of data comes in is larger than the number of those fetched out from the buffer. The rule-based solution is also not efficient enough to handle the incoming data for some large

speedup factors, therefore, we observe its latency increasing. However, since the generator already reaches its maximum capacity at the speedup factor 50, the latency of the rule-based solution becomes stabilized after the point, too.

6. DISCUSSION

In the previous section, we evaluated our implementation methods with the performance-related criteria. In this section, we discuss their total quality for an event-monitoring system by taking non-performance measures into consideration as well. The selected non-performance measurements are the implementation effort, maintainability, extensibility and optimization. The first three criteria were adopted from [1]. Although the evaluation criteria reflect our team's subjective perspective, it shows an interesting implication about the trade-off between the performance and the efficiency in implementation and maintenance of event-based systems.

Implementation effort: This measures the time for developers to complete the program. It includes the level of required programming skill, previous knowledge about external APIs, and setup and integration efforts with the existing rule engine platform.

Maintainability: It is related with the easiness of inserting codes to handle new event stream and modifying current codes to change the operation for current events. The modular structure can support this requirement well.

Extensibility: It gauges whether the solution provides general features for distributed event handling like multi-threading. These features help extend our solution to fit into a large and complex system.

Optimization: It probes how much the global optimization is applied to accelerate the processing speed. The factor is especially important in such a problem that requires to process a huge amount of input stream on the real-time basis.

The summary that shows the comparison results of our solutions in the above criteria is presented in Table 1.

Table 1. Comparison among our solutions

	Custom solution	**Rule-based solution**	**CEP solution**
Implementation effort	Expert programming knowledge	Average programming knowledge + external API	Easy + external API
Maintainability	Difficult	Custom functions required	Easy
Extensibility	Not supported	Ready-made distributed processing features available	
Optimization	Yes	Partly (event timeout handling)	No

All the solutions that we explored showed different pros and cons. Generally speaking, the optimization took the most important role to improve the performance. Although the CEP solution was easy to implement, it resulted in the worst performance. The rule-based solution required more engineering knowledge and development effort, but it gave much better results than the CEP-based solution. Finally, the optimized solution in C++ could achieve insignificant latency and throughput as large as the generator. In conclusion, we found throughout our experiments that the existing out-of-box solutions provided convenient tools to implement a event-based system, but needed additional efforts to improve performance. The custom solution is usually much difficult to implement, maintain and change, but can enjoy the fruits of the most advanced optimization. Therefore, the technical selection among different solutions in real-world systems needs to take both performance and non-performance requirements into consideration.

7. REFERENCES

[1] Aders, L., Buffat, R., Chothia, Z., Wetter, M., Balkesen, C., Fischer, P. M. and Tatbul, N. 2011. *DEBS'11 Grand Challenge: Streams, Rules, or a Custom Solution?* Technical Report #749. ETH Zurich.

[2] Allen, J. F. 1981. An interval-based representation of temporal knowledge. In *Proceedings of the 7th international joint conference on Artificial intelligence* (Vancouver, BC, Canada, August 24-28, 1981). IJCAI'81. Morgan Kaufmann Publishers Inc., San Francisco, CA, 221-226.

[3] Cormode, G., Muthukrishnan, S. and Yi, K. 2011. Algorithms for distributed functional monitoring. *ACM Transactions on Algorithms*. 7, 2 (March 2011), Article 21. Preliminary version in SODA'08. DOI= http://dx.doi.org/10.1145/1921659.1921667

[4] DEBS2012 Grand Challenge. http://www.csw.inf.fu-berlin.de/debs2012/grandchallenge.html.

[5] Forgy, C. L. 1982. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*. 19, 1 (September 1982), 17-37. DOI= http://dx.doi.org/10.1016/0004-3702(82)90020-0

[6] JBoss Drools - The Business Logic Integration Platform. http://www.jboss.org/drools.

[7] JBoss Drools Fusion Manual, http://docs.jboss.org/drools/release/5.4.0.CR1/drools-fusion-docs/pdf/drools-fusion-docs.pdf

[8] Ley, E. D. and Jacobs, D. 2011. Rules-based analysis with JBoss Drools: adding intelligence to automation. In *Proceedings of the 13th International Conference on Accelerator and Large Experimental Physics Control Systems* (Grenoble, France, October 10-14, 2011).

[9] Stonebraker, M., Cetintemel, U. and Zdonik, S. 2005. The 8 requirements of real-time stream processing. *SIGMOD Record* 34, 4 (December 2005), 42-47. DOI=http://dx.doi.org/10.1145/1107499.1107504

[10] Walzer, K., Breddin, T., and Groch, M. 2008. Relative temporal constraints in the Rete algorithm for complex event detection. In *Proceedings of the second international conference on Distributed event-based systems* (Rome, Italy, June 1-4, 2008). DEBS '08. ACM, New York, NY, USA, 147-155. DOI=http://doi.acm.org/10.1145/1385989.1386008