DEBS Grand Challenge: Odysseus as Platform to Solve Grand Challenges

Dennis Geesen
University of Oldenburg
Department of Computing Science
Oldenburg, Germany
dennis.geesen@uni-oldenburg.de

Marco Grawunder
University of Oldenburg
Department of Computing Science
Oldenburg, Germany
marco.grawunder@uni-oldenburg.de

ABSTRACT

This paper provides the description of our solution that we made for the DEBS'12 Grand Challenge. We used the Odysseus data stream management framework to solve the given problem of monitoring large hi-tech manufacturing equipment. For that, we show how the challenge is modeled in our terms, how we solved several problems and discusses the effort and benefits of our approach.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Management

Keywords

Complex Events, Data Stream Processing, Grand Challenge Solution

1. INTRODUCTION

The grand challenge is a competition that is provided by the 6th ACM International Conference on Distributed Event-Based Systems (DEBS 2012). The goal is to implement a provided problem, whose solutions are evaluated on correctness, throughput and latency. The problem is described by two queries that are used to monitor large hi-tech manufacturing equipment. The first query aims to monitor the duration between an additive sense and a corresponding valve release. The second query is used to monitor the power consumption. We solved the problem by using the data stream framework called Odysseus [1]. It uses a stream-based variant of the relational algebra, which is a well-known and common concept in relational data base systems to provide semantically correct and deterministic solutions.

In this paper, we show how we solved the grand challenge. We first describe in Section 2 the two generated queries and show how they reflect the proposed queries. Afterwards, we give an outlook in Section 3 over the benefits and efforts that come with our system. Finally, we conclude the paper in Section 4

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DEBS '12, July 16–20, 2012, Berlin, Germany.
Copyright 2012 ACM 978-1-4503-1315-5 ...$15.00.

2. THE SOLUTION

To solve the challenge, we use the data stream management framework (DSMF) Odysseus[1] , which is designed to build several kinds of data stream management systems (DSMS), e.g. domain-specific solutions, for rapid prototyping or evaluation of new techniques. The adaptability is primarily provided by so-called variation points in conjunction with a component-based architecture. A variation point is typically an interface that can be extended or customized is normally implemented by components. Thus, there are several components which can be combined via a configuration to get concrete DSMS. An example is the kind of data that can be processed (e.g. relational, RDF, etc.) or the adaption of the query processing (e.g. the query language or the optimizer). Since the given data is relational, we used a relational configuration as an initial system. Additionally, we also use the time-interval component that is needed to handle the temporal context by annotating a validity interval to each incoming event. This component is also responsible for a memory-aware processing of unbounded data streams and additionally for a deterministic and semantically correct processing. The relational and interval based processing of events is provided via a stream algebra [4].

The stream algebra is a set of relational operators as they can also be found in most relational database systems. The major difference is the consideration of time-intervals for the validity of each event, so that the system only processes events that temporally correlate. Thus, the system can use the time-intervals to divide an unbounded data stream into portions, which prevents the system to be endlessly blocked or to get a memory overflow. The system already provides a basic set of algebra operators, e. g. for selections, projections, aggregations, unions, intersections or joins. The consecutive connection of such algebra operators produces a directed graph—the so-called query plan—where the algebra operators are nodes and the data flow are directed edges. Each event gets at a source operator into the system and flows through the whole query plan until the event leaves the system at a sink operator. Each operator that is passed by an event executes its specific operation on the event and forwards it to the next algebra operator in the query plan.

[1] http:\\odysseus.offis.uni-oldenburg.de

Since Odysseus allows the integration of user-defined algebra operators, the concept is very close to event processing agents like they are proposed in [2]. Accordingly, their combination to a query plan can be compared to an event processing network and the consideration of time-interval is mentioned as temporal context in [2]. In the following, we show how we solved the grand challenged by using the stream algebra. For that, we separately describe the solution for each query.

2.1 Query 1

The first query aims to monitor the duration of changes for a Chem Additive sensor and its corresponding valve. There are three pairs of sensor and valves and two tasks for each pair. One task is supposed to produce an alarm if the duration increases more than one percent and another task is the calculation of a trend via a regression. Since all three pairs only vary in the observed Chem, we only explain one pair and path of the query plan. This path is shown in Figure 1. The first four algebra operators (see *Source* in Figure 1) are used for the connection to the data server. The `Receiver` opens a connection and deserializes each incoming event from the Google Protocol Buffer into an internal representation. The `MetadataCreation` and `MetadataUpdate` operators annotate each event with a time-interval, which is given by the timestamp from attribute `ts`. The next part, which is shown as *Part 1* in Figure 1, represents the operators one and four of the given query. The `Project` reduces the attributes to `pp04` or `pp05` respectively. The `ChangeDetect` is an operator that only produces a new event, if the value of the incoming event is different from the value of the previous event. Thus, it detect changes from 0 to 1 or from 1 to 0 and forwards the last incoming event to the next operators, which are shown in Figure 1 as *Part 2*. The results of the `ChangeDetect`s go into a `PatternDetect` operator, which is equal to *operator 7*. A `PatternDetect` operator gives the possibility to match complex event patterns by implementing SASE+ (see [3] for a detailed description). Therefore, we use the `PatternDetect` operator for matching a sequence of two events from `s05` and `s08` where both values for `edge` are equal. The results of the `PatternDetect` operator are the two timestamps `s05.ts` and `s08.ts` from the two events.

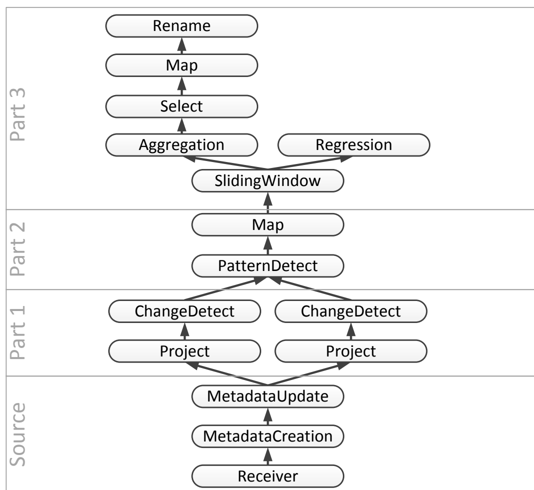

Figure 1: Overview of query 1

The subsequent `Map` is responsible for calculating the duration `dt` from `s05.ts` and `s08.ts`. The last part of this query is visualized in Figure 1 *Part 3* and shows *operator 10* and *operator 11* for this path. Both operators focus on the last 24 hours, so that we firstly use a `SlidingWindow` that sets the validity of each event through its time interval. The system recognizes this validity in each following operator by processing only simultaneously valid events together. Therefore, the `Aggregation` calculates the minimum and maximum of the duration `dt`. The validity time interval, which is annotated as metadata to each event, is used by the aggregation for considering only the last 24 hours. The minimum *min* and maximum *max* are used to calculate the difference of any two values and if they increase more than 1%. This calculation is done by a `Select` that only forwards events, if they fulfill the predicate $max/1.01 > min$. We consciously calculate from *max* as the basis, because *min* might be zero, so that 1% cannot be correctly calculated. Since the `Select` only forwards events, we need the `Map` to transform them into events with `alarm=1`. The second branch after the `SlidingWindow` only holds the `Regression`. The regression uses the duration `ts` for the x-value and the timestamp `dt` for the y-value and calculates a linear function using the method of least squares. Therefore, the `Regression` produces events with attributes `slope` and `intercept`, reflecting a function $f(x) = slope \cdot x + intercept$ where x might be a timestamp. These values are also used to visualize the function.

2.2 Query 2

The second query is used for monitoring the energy consumption. The first part of the query plan is the same as in query 1, so that Figure 2 shows only the next parts. *Part 1* reflects *operator 1*, *2* and *3* from the given query. First, the `PeriodicWindow` calculates a window of one second over the projected stream that slides each second and annotates each event with validity. The aggregation, which is calculated three times for `mf01`, `mf02` and `mf03`, is combined into one `Aggregation` operator[2]. The result is an event that holds the average, maximum and minimum for each of these three attributes. These values are used by the subsequent `Map` for calculating the relative variations. Therefore, against the proposed query, we have only one intermediate data stream. Afterwards, the `MetadataUpdate` resets the validity time-intervals, because the system is normally designed

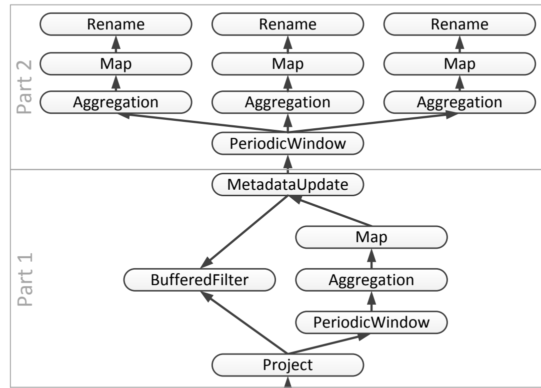

Figure 2: Overview of query 2

[2]else there would have been a temporal join necessary

for only one window operator per query, but in that case, we have two different windows for one query, e.g. in *operator 1* and *operator 5*. *Part 1* in Figure 2 also shows *operator 4*, which is implemented via the `BufferedFilter`. This operator has two inputs and one predicate. One input stream is always buffered for a period of time (in our case 20 seconds), while the `BufferedFilter` evaluates the predicate over the second input stream. If an event fulfills the predicate, the `BufferedFilter` delivers the whole buffer and additionally further events from the first input as long as an event does not fulfill the predicate. If an event does not fulfill the predicate for another period of time (in our case 70 seconds, the `BufferedFilter` stops to forward further events. Furthermore, the `BufferedFilter` enriches each forwarded event with the event that fulfilled the predicate. Since the `BufferedFilter` only forwards the incoming stream and we only want to have `mf01`, `mf02`, `mf03` and `ts` from the source, so that a `Project` is needed at the beginning of *Part 1* to reduce the attributes. Notice, that this is not necessary for the other operators.

Since the last three operators (*operator 5, 6* and *7*) are equal except for the considered attribute, we only explain one sub path. The `PeriodicWindow` realizes the tumbling window for 60 second, so it keeps 60 seconds and outputs every 60 seconds. The output is forwarded to an `Aggregation`, which calculates the maximum of the timestamp `ts` (or in our case the last one of the 60 seconds window). Afterwards, the `Map` calculates the cubic root of `avg-mf01`, `avg-mf02` and `avg-mf03`. The `Rename` finally renames the result of the `Map` to `ts` and `pwr`.

2.3 Visualization and Interface

Odysseus provides a graphical user interface that is based on the Eclipse Rich Client Platform (RCP) called Odysseus Studio. The Studio provides several mechanisms for controlling, monitoring and visualization. It is possible to create projects and, e. g., script files. These script files can be used to define queries (like in listing 1). Furthermore, it is possible to show all installed sources and sinks or current installed queries. Besides the demand visualization of the trend for query 1, there are a lot of other visualizations like bar, line or pie charts or simple tables.

2.4 Experiments

We made different experiments to determine latency and throughput of the queries. We run the experiments on an AMD Phenom II X6 1090 T with 3,2 GHz and 16 GB of main memory and used the provided generator at an acceleration speed of 1000.

To calculate the latency we needed to switch to another Odysseus processing configuration, where latency information is attached to each processed element. A start timestamp is set automatically when the event enters the system (by the `MetadataUpdate` operator). A special operator `LatencyCalculationPipe` is used to set the end timestamp before the tuple leaves the system. The difference between these two timestamps is the latency in nano seconds. Note, if more than one event is needed to create an output, the latency information of the last participating event is used. E.g. in the `ChangeDetect` operator the event that completes the matching provides the latency meta data. Odysseus provides meta data merge functions, which can be attached to the operators. No operator needs to be changed to handle latency correctly. With this preparation we can measure for each sink the latency.

The results for Query 1 can be found in Table 1. Figure 3 shows the results for the alarms, where all three lines for operator 10, 12 and 14 are quite similar and alternate about one second. This uniform alternating behavior is due to the aggregation processing, because it has to wait until it is certain that it has not missed any value for the maximum or minimum calculation. Since there are only few intermediate results after the `ChangeDetect` and the `PatternDetect`, there are not enough events for the aggregation to determine if it has everything considered. Thus, the aggregation has to wait and cannot write out something, although there are finished results. The results from the regressions, which becomes the same data from the window operator, in Figure 4 emphasize this behavior, because the regression does not have to wait, so that their latencies are very smaller and their variations are not uniform like the results from the aggregation. Although it is possible to produce a lot of small results with a small validity from the aggregation instead of

sink	latency in ms		
	mean	min	max
operator 10	1419.357	80.231	14348.673
operator 12	1631.492	55.851	39455.618
operator 14	1514.277	57.746	20870.994
operator 11	2.727	0.907	14.956
operator 13	2.054	0.357	9.340
operator 15	3.533	0.828	12.614

Table 1: Experimental results of query 1

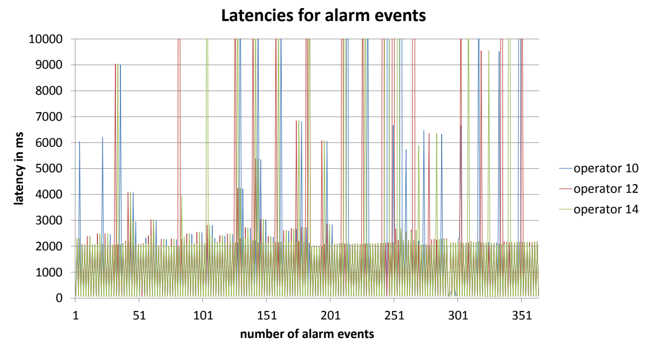

Figure 3: Query 1 latencies for ops. 10, 12 and 14

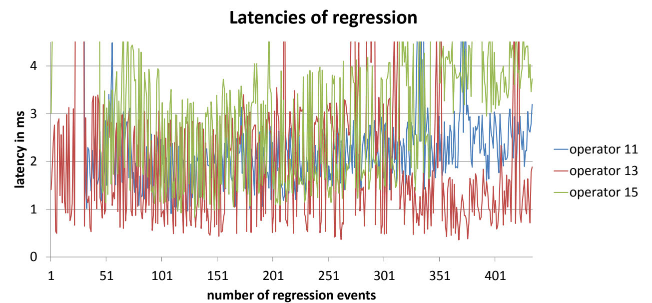

Figure 4: Query 1 latencies for ops. 11, 13 and 15

one with the whole validity and this would produce significant smaller latencies, we hold the view that our solution is more reproducible and does not imitate wrong latencies.

The results for Query 2 can be found in Table 2. Since we observed a drift in the results of operator 4 (see Figure 5 for one violation), we split these data into two parts. The first part shows the latency of all *buffered events* for 20 seconds before the violation and the second part shows the 70 seconds of *further events* after the violation. Since the first part is already buffered, the events only wait and there is nothing to do, they can be immediately forwarded if the violation occurs, thus, the latency is relatively small. The second part is forwarded after the violation and therefore has to be processed while new events arrive, so that there is a higher latency. Since there is a frequency of the raw data about 100Hz, the first part exists of about 2000 and second part exists of about 7000 events. Furthermore, the latencies for the power consumption (operator 5 to 7) are shown in Figure 6. The latencies alternate around their mean. Although there is an order in the latencies (s5 at the bottom, s6 in the middle and s7 at the top), this is just due to the order of the scheduler. Thus, if operator 7 (s7) would be scheduled first, it would have the smallest latency.

sink	latency in ms		
	mean	min	max
operator 4 (buffered events)	0.064	0.055	0.237
operator 4 (further events)	0.190	0.171	2.244
operator 4 (all events)	0.162	0.055	2.244
operator 5 (s5)	0.332	0.217	0.548
operator 6 (s6)	0.486	0.344	28.519
operator 7 (s7)	0.631	0.385	28.601

Table 2: Experimental results of query 2

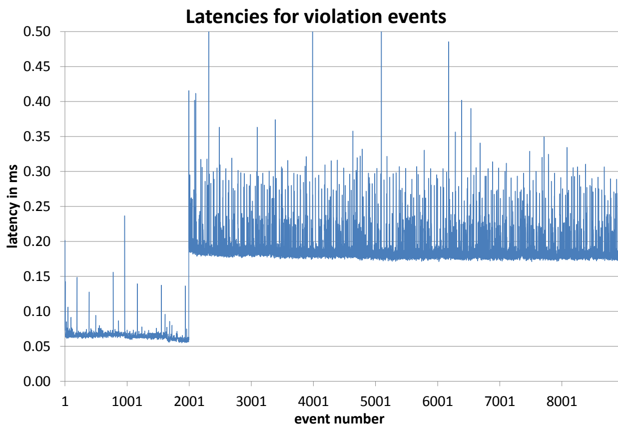

Figure 5: Query 2 latencies for operator 4

We defined throughput as the number of elements per time that the system can receive. By this, we do not have to consider the different filter in the system. One problem was that the provided input was not uniformly distributed. There were some ”time gaps”, sometimes more than 8 days. Ignoring these gaps (and treating only the first 16 million elements) we had an average throughput of about 6600 events per second for query 1 and 7200 events for query 2.

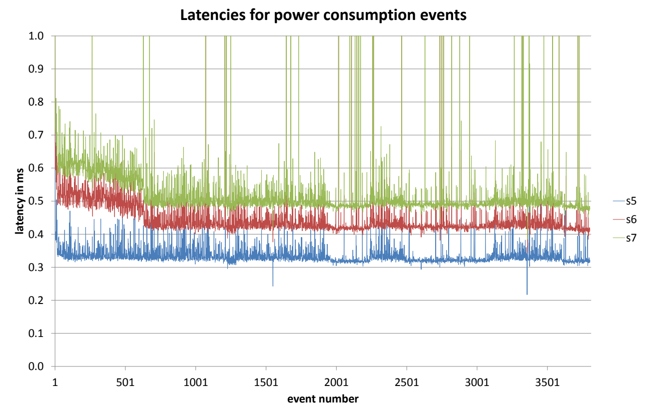

Figure 6: Query 2 latencies for ops. 5, 6 and 7

3. EFFORT, BENEFIT AND ADVANTAGES

This section describes the efforts and benefits that our system provides. It should also demonstrate how easily we could solve some problems and why we can assume that our solution is semantically correct and delivers deterministic results.

3.1 Declarative Language

On benefit of Odysseus is a query interface that supports a various set of languages. In that case, we used a procedural query language (PQL) which allows one to specify the order of all stream algebra operators. Listing 1 shows an excerpt of the formulated query for *operator 1* and *2*.

Listing 1: Excerpt of query 1

```
#PARSER PQL
#QUERY
/// Operator 1
s05 = RENAME({type='s05', aliases = ['ts', 'edge']},
        CHANGEDETECT({attr=['pp04'],
            heartbeatrate=100},
          PROJECT({attributes=['ts','pp04']},
            gchSource)
        )
      )
...
```

As you can see, the defined script allows among other things the possibility to switch the parser, e.g. to a SQL-like query language. Furthermore, the declarative formulation in PQL has the advantage that we can easily switch operators or change parameters like predicates or the number of projected attributes. e. g., the attributes `bm05` or `bm08` were changed during the challenge and we only had to change the attribute-names in the formulated PQL query. Furthermore, the declarative approach allows one to concentrate on the query and not on specific algorithms for projecting, joining or selecting raw events.

3.2 Correctness and Determinism

One of the most important key features of Odysseus is the usage of a fixed defined algebra for streaming purposes. This stream algebra is based on time-intervals introduced by [4] and defines a set of algebra operators, which are adapted from traditional database systems. The main idea is to cut each step (or change) of the streaming query plan into a snapshot, so that we have a sequence of snapshots. Each snapshot can be seen as a normal state of a static database

where only events are considered whose time-intervals are valid during the snapshot. Therefore, the concept of time-intervals realizes the temporal context of events and it makes sure that, e. g., the aggregation in query 2 (*operator 1*) only considers events for a snapshot of one second. Against solutions where these snapshots are implemented in each operator, our solution solves this by annotating the events with a time-interval. So, our solution only recognizes the application time and not system time or event mixes them. This has the advantage that there could be no race conditions and it is possible to run the data generator in even higher frequencies without changing the results, because the results are always calculated with the help of the application time (or in our case `ts`) in the same way. In reference to the challenge, our solution would produce always the same alarms or the same power consumption for the same input data and no data is lost due to wrong calculations. Since the provided data file has several space between two measurements (e.g. 2012-02-27 01:55:33 follows directly 2012-02-24 13:32:19), the application time approach was also very helpful, because—in reference to the 24 hour window—we do not have to wait during this space of time. Oppositely, this holds even if the sensor or the communication of the manufacturing equipment produces bursts or delays. However, the processing of Odysseus strongly depends on the order of the start timestamps. There were also some wrong ordered timestamps in the provided data file (e.g. 2012-02-23 13:12:01 and after this 2012-02-23 12:08:22), so that we have to omit all events that are out of order. Although there are concept for an out of order processing, we do not use this, because we assume that this wrong order comes from the concatenation of the recordings and would not happen in a real manufacturing monitoring.

Reproducible results are another advantage of the defined algebra, because every calculation is defined in advance and can be computed by hand. Therefore, our solution will produce the same results for the same incoming data. This is advantageous when, e.g., *operator 4* in *query 2* raises the predicate condition and produces an output. Then, an engineer can reproduce how and which raw data causes the output.

3.3 Built-in Optimizations

A third advantage of the relational stream algebra is the adaption from traditional database systems. If a stream algebra operator leads to the same result like the sequence of all snapshots on which the non-streaming counterpart was executed, the operator is called snapshot-reducible (see [4]). Each streaming operator that is snapshot-reducible has the same properties of its non-streaming counterpart. This allows—among other things—the use of existing rules from traditional databases for deleting, swapping or moving particular operators without any loss of semantic. Thus, the system automatically optimize the query plan by deleting unnecessary operators or pushes selections down to the source, so that computationally intensive operators are pushed up and have to process muss less events. Since these optimizations are based on the traditional relational algebra, it cannot be applied to all other stream operators like the `PatternDetect` operator. Such operators are static in the plan and divide the query plan into optimizable parts of relational algebra operators, so we were still able to use this technique for the challenge.

A second optimization we used was query sharing, which reuses already running operators, so that the new query plan is merged into existing one and the whole system load is reduced. This optimization was, e. g., automatically done for the sources (see Figure 1 and Figure 2), because this part is used in both queries. We also used this concept by hand, because we combined *operator 1, 2, 3* of *query 2* into one.

Besides the previous two static optimizations during the deployment of new queries, there are also dynamic optimizations during the execution. Since the system does not know the exact characteristics of an incoming stream, it can observe its behavior and, e. g., reorganize the query plan at runtime if maybe another operator order would be better. However, we do not use these optimization techniques, because the system has to maintain statics for only some operators that do not promise any possibilities for a dynamic optimization.

So-called heartbeats are another optimization. Since some operators have to wait for a successive event until they can produce a result, they are blocked, because a result cannot be produced until the operator is sure that it considered all events within the results validity. This validity is given by the chronological sequence of the data stream, so that the operator can use the validity time-intervals of the events to decide whether it has seen all needed events for a result. But, if there are no incoming events for a long time, the operator blocks. At this point, heartbeats indicate the chronological progress of the data stream, because a heartbeat means that there is no event in the future with a timestamp in the validity time-interval that is greater than the timestamp of the heartbeat. Thus, the operator can use this information from the heartbeat to produce results earlier than before. Therefore, heartbeats could be seen as a processing optimization. We used the heartbeat mechanism to unblock several operators. Especially the aggregations (e.g. *operator 1-3* of *query 2*) have to wait for an event until it knows that it calculates the average with all events within the second. The heartbeat mechanism makes sure that the aggregation does not have to wait for event if it receives a heartbeat. Thus, this technique improves the latency.

3.4 Extensibility

As mentioned before, Odysseus is a framework that is written in Java and provides a various number of components, which can be extended or configured for specialized solutions. Among the possibility to define and plug in new query languages, new processing rules or scheduling algorithms, one major extension point is the insertion of new algebra operators. Since our system had no `ChangeDetect` or `BufferedFilter` before, we implemented and integrated them into the system. To extend Odysseus with a new operator, there have to be done at least three steps. First, a logical operator has to be defined by extending an abstract class. This logical part defines *what* the operator does, what kind of schema (the list of attributes) it needs and provides and what parameters are needed. In this case, a logical `ChangeDetect` describes that the input is equal to the output schema. Furthermore, it defines the attribute that is considered by the change detection. The logical operator can be annotated with a special name, so that it is automatically provided as an allowed operator in PQL (e.g. like the `ChangeDetect` for query 1 in Listing 1). The second step is to define `how` the operator works by implementing a physi-

cal operator. Primarily there is only one processing-function that has to be implemented. The physical `ChangeDetect`, e. g., only checks in this function, if the defined attribute of the current event differs from the last event. If they are different, a provided transfer-function can be used to forward the event to all subsequent operators. The third step is to define a transformation rule, which is used to transform the logical operator into its physical counterpart. Notice, there are a lot of templates to minimize the time of implementation.

The separation of logical and physical operators is adapted from common database systems and has two major advantages. For the one hand, there could be more than one physical implementation for one logical operator, so that an optimizer can choose the best implementation for different queries. Due to the extensibility of Odysseus, new physical implementations for existing operators can be integrated easily. On the other hand, the logical operator is not fixed for a special data type. Thus, additional transformation rules for one logical operator can be used to choose another physical operator, e. g. for another data type than relational events. Notice, the optimization is still the same, because it is based upon the logical operators.

The extensibility was very useful for solving the challenge, because we were able to integrate new operators very easily. The processing steps of a `ChangeDetect`, e. g., could alternatively be implemented via existing stream operators from the relational algebra. However, it was easier for us to integrate a new operator than defining its semantic via traditional stream operators. Furthermore, such a traditional solution would be too computationally intensive in this scenario, so a new operator induces also a better load and better latencies.

4. CONCLUSION

The solution of the grand challenge of the DEBS 2012, we presented in this paper, was done on the basis of Odysseus. Odysseus is a data stream management framework, which allows various possibilities for extensions and configurations. An already existing configuration for relational events in consideration of the temporal context using validity time-intervals was used as an initial system. We used the procedural and declarative query language to formulate the given queries and implemented two missing operators, which could be easily integrated into the system. Finally, we showed some key features of Odysseus to give an idea how much effort was needed and how much advantages and benefits is brought by the system. To sum up, these are:

- The use of time-intervals as the stream model allows a semantically defined, deterministic, system time independent and reproducible processing of the manufacturing data, which is insusceptible against race-conditions or bursts e.g. caused by broken communications or sensors.
- The optimization techniques like query sharing or restructuring allows the reduction of load and latencies without any loss of semantics, so that e.g. alarms are produced faster.
- The architecture of Odysseus is designed to be very extensible and changeable, so that later changes of query definitions can be applied very easily. Additionally, new requirements for monitoring other manufacturing equipment could be also inserted by defining new PQL-queries and/or implementing new operators if needed.

Our local experiments with the given data showed that the system runs smoothly, because it uses a fixed amount of resources like memory and processing time. The latency and throughput was like expected, so that query 2, e. g., produced each minute a new result for the power monitoring.

5. REFERENCES

[1] H.-J. Appelrath, D. Geesen, M. Grawunder, T. Michelsen, and D. Nicklas. Odysseus - a highly customizable framework for creating efficient event stream management systems. In *Distributed Event Based Systems*. ACM, 2012.

[2] O. Etzion and P. Niblett. *Event Processing in Action*. Manning Publications, 2011.

[3] D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immerman. On supporting kleene closure over event streams. *2008 IEEE 24th International Conference on Data Engineering*, pages 1391–1393, Apr 2008.

[4] J. Krämer and B. Seeger. Semantics and implementation of continuous sliding window queries over data streams. *ACM Transactions on Database Systems*, 34(1):1–49, Apr 2009.