Model-based Validation of Streaming Data

(Industry Article)

Cheng Xu
Department of Information Technology
Uppsala University
Box 337, SE-75105, Sweden
+46 18 471 7345
cheng.xu@it.uu.se

Daniel Wedlund
AB Sandvik Coromant
R&D Application solutions,
Functional Products
SE-811 81 Sandviken, Sweden
daniel.wedlund@sandvik.com

Martin Helgoson
AB Sandvik Coromant
R&D Application solutions,
Functional Products
SE-811 81 Sandviken, Sweden
martin.helgoson@sandvik.com

Tore Risch
Department of Information Technology
Uppsala University
Box 337, SE-75105
+46 18 471 6342
tore.risch@it.uu.se

ABSTRACT

An approach is developed where functions are used in a data stream management system to continuously validate data streaming from industrial equipment based on mathematical models of the expected behavior of the equipment. The models are expressed declaratively using a data stream query language. To validate and detect abnormality in data streams, a model can be defined either as an analytical model in terms of functions over sensor measurements or be based on learning a statistical model of the expected behavior of the streams during training sessions. It is shown how parallel data stream processing enables equipment validation based on expensive models while scaling the number of sensor streams without causing increasing delays. The paper presents two demonstrators based on industrial cases and scenarios where the approach has been implemented.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems – *Parallel databases*, *Query processing*

Keywords

Equipment Monitoring; data stream management system; data stream validation; parallelization; anomaly detection.

1. INTRODUCTION

Emerging business scenarios such as provision of total care products, product service systems (PSS), industrial product-service-systems (IPS^2), and functional products [3] [4] [5] [14] [15] imply needs to efficiently monitor, verify and validate the functionality of a delivered product in use. This can be done with regard to pre-defined criteria, e.g. productivity, reliability, sustainability, and quality. A functional product in this context mean an integrated provision of hardware, software and services.

DEBS'13, June 29–July 3, 2013, Arlington, Texas, USA.

In case of machining several factors and dependencies have to be considered, which in turn means that data (e.g. in-process data) from the machining process, information (e.g. about cutting tools), and knowledge (e.g. physical models), from several domains have to be captured, combined and analyzed in a comprehensive knowledge integration framework that includes quality assurance of data, validation of models, learning capabilities, and verification of functionality [10]. A considerable challenge is to scale up data analysis for handling huge amount of equipment.

In this context novel software technologies are needed to efficiently process and analyze the large data streams, in particular related to in-process activities, and to facilitate the steps towards increased automation of the related processes.

In manufacturing industry, equipment such as machine controllers and various sensors are installed. This equipment measure and generate data during the machining process, i.e. in-process. Depending on the case a huge amount of parameters (e.g. power, torque, etc.) at different sample rates (ranging from a couple of HZ to 20 kHZ) need to be processed. Processing data streams from controllers and sensors is critical for monitoring the functional product in use. For instance, the parameters related to the power consumption could help the engineers to analyze the process, compare different application strategies, monitor and maintain the hardware e.g. to get an indication of the degree of tool-wear or when a tool needs to be replaced or machine maintenance is required.

Often a mathematical model of the process can be developed, e.g. to calculate the expected power consumption and detect abnormal behavior. In other cases, when there is no such model pre-defined, a model can be learned based on observing sensor readings during training sessions. This requires a general approach to define the correct behavior of the equipment either analytically or statistically.

Data Stream Management Systems (DSMSs), such as Aurora [1] and STREAM [16], process continuous queries, CQs, over data streams that filter, analyze, and transform the data streams. A simple CQ can be: "give me the sensor id and the power consumption whenever the power is greater than 100W". However, detecting abnormal behavior in equipment often involves advanced computations based on knowledge about the machining process, e.g. theoretical models of the equipment's behavior, rather than just simple numerical comparisons in a

query condition. An advanced CQ could be: "given a power consumption model computing the theoretical expected power consumption at any point in time, give me the sensor id whenever the difference between the actual power consumption and the theoretical expected power on the average is greater than 10W during 1 second."

To enable general stream validation based on mathematical models, the system called SVALI (Stream VALIdator) was developed and used in industrial applications. The system provides the following facilities:

1. Users can define and install their own *analytical* models inside the DSMS to validate correct behavior of the data streams. The models are expressed as side-effect free functions (formulae) over streamed data values.
2. For applications where no theoretical model can be easily defined, the system can also dynamically learn a model based on some existing observed correct behavior and then use that *learned* model for subsequent validation.

SVALI is a distributed DSMS extending SCSQ [22] with validation functionality. Many SVALI nodes can be started on different compute nodes. The distributed SVALI architecture enables processing of validations in parallel without causing unacceptable delays by the often expensive computations, as shown in this paper.

The paper is organized as follows. In Section 2 the architecture of a SVALI node is presented. Section 3 presents two general strategies for stream validations in SVALI called *model-and-validate* and *learn-and-validate,* illustrated by real-life industrial examples. Section 4 presents results from simulations on how the parallelization enables scalable processing of expensive validation functions in the applications, and Section 5 discusses related work. Finally, Section 6 summarizes and outlines future work.

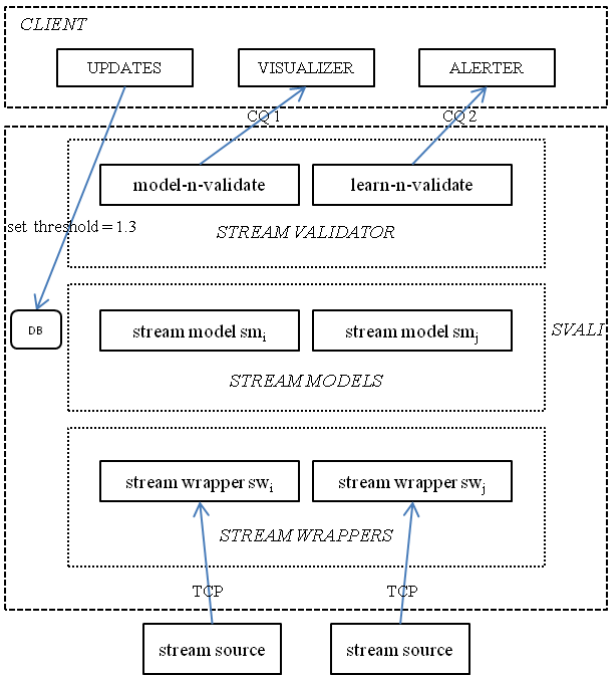

Figure 1. System Architecture

2. SYSTEM ARCHITECTURE

Figure 1 shows the architecture of the SVALI system. Different kinds of data streams are collected from *stream sources* of sensor readings. SVALI is an extensible DSMS where new kinds of stream sources can be plugged in by defining *stream wrappers*. A stream wrapper is an interface that continuously reads a raw data stream and emits the read events to the SVALI kernel. On top of the stream wrappers, equipment specific *stream models* over raw data streams analyze the received stream tuples to validate that different kinds of equipment behave correctly. A stream model is a function over either individual stream tuples or over windows of stream tuples. Stream models are passed as a parameter to the *stream validator* that applies the models to produce *validation streams* where deviations from correct behavior are indicated.

The main-memory *local database* stores meta-data about the streams such as stream models, tolerance thresholds, collected statistics, etc.

For validating streaming data using an analytical stream model the system provides a second order function, called *model_n_validate()*. It validates data streaming from sensors on a set of machines based on a stream model function and emits a *validation stream* of significant deviations for malfunctioning machines.

It is also possible to automatically build at run-time a model of correct behavior based on observed correct streaming data using another second order function called *learn_n_validate()*. During a learning phase it computes statistics of correct behavior of the monitored equipment based on a user provided statistical model. After the learning phase the collected statistics is stored in the local database and used as the reference with which the streaming data will be compared. As for model-and-validate, the system will emit a validation stream when significant deviations from normal behavior are detected.

The validation streams can be used in CQs. For example, in Figure 1 CQ1 is used as input to a visualizer of incorrect power consumption and CQ2 is a stream of alert messages signaling abnormal power consumption.

It is possible to dynamically modify the validation models while a validating CQ is running by sending update commands from the application to the local database. For instance, it is possible to change some threshold parameter used in an analytical model for a particular kind of machine, which will immediately change the behavior of the running validation function.

Usually the process of validation of a single machine's behavior depends on data streaming only from that particular machine combined with data in the local database. The overall detected abnormal behaviors are then collected by merging the individual machines' validation streams. For such CQs, the system automatically parallelizes the execution so that each compute node executes validation functions for a single data stream source independent of streams from other machinery. The system then merges the validation streams before delivering the result to the application. All the nodes contain the same database schema of machine installations and meta-data such as thresholds used in validation models. In the paper it is shown that this parallelization strategy outperforms validation on a single node and enables the delay caused by the monitoring of many machines to be bounded.

3. MODEL BASED VALIDATION

The functionalities of the two kinds of model based validation methods in SVALI are described along with examples of how they are applied on industrial equipment in use.

3.1 Model-and-validate

When the expected value can be estimated based on an analytical stream model, it is defined as a function which is passed as a parameter to the general second order function called *model_n_validate()* that has the following signature:

```
model_n_validate (Bag of Stream s, Function modelfn,
                  Function validatefn)
    -> Stream of (Number ts, Object m, Object x)
```

The user defined stream model function, *modelfn(Object e)->Object x,* specifies how to compute the expected value *x* based on a stream element *e*. A stream element can be a single stream tuple or a window of tuples.

The user defined validation function, *validatefn(Object r, Object x)->(Number ts, Object m),* specifies whether a received stream element *r* is invalid compared to the expected value *x* as computed by the model function. In case *r* is invalid the validation function returns the *ts* time stamped invalid measurement *m* in *r*.

The element of the validation stream returned by *model-and-validate()* are tuples *(ts, m, x),* where *ts* and *m* are computed by the validation function and *x* is computed by the model function.

CQs specification involving model-and-validate calls are sent to a SVALI server as a text string for dynamic execution. It is up to the SVALI server to determine how to execute the CQs in an efficient way. In particular SVALI transparently parallelizes the execution to minimize the delay caused by executing expensive validation functions.

3.1.1 Demonstrator 1

This section demonstrates how model-and-validate is used to validate the power consumption in an industrial case based on a milling scenario. The case, including the framework, meta-data, models, a cutting tool, a machine tool, related equipment to capture the needed in-process data, and a stream server called Corenet was provided by Sandvik Coromant. The streaming process data used in this demonstrator was simulated using real recorded process data from Sandvik Coromant. To be specific, the data was collected from a MoriSeiki 5000 with a Fanuc control system that was equipped with the Kistler sensor system 9255B, and DMG with a Siemens control system. The difference between running Corenet with a recorded file compared to Corenet with connection to a machine is just a matter of configuration. In this paper a consistent behavior was needed to evaluate the performance and therefore it was of benefit to use recorded data from earlier machining attempts.

Figure 2 illustrates how the milling process was performed. The parameters in Table 1 describe the milling process. Tool working engagement is denoted by a_e feed per tooth by f_z, maximum chip thickness by h_{ex}, cutting depth by a_p cutting speed by v_c and the number of cutting edges by z_c.

Figure 2. The side milling process

Table 1. Parameters that measured

a_e [*mm*]	f_z [*mm/tooth*]	h_{ex} [*mm*]	a_p [*mm*]	v_c [*m/min*]	z_c
2	0.0756	0.05	20	200	4
3	0.0641	0.05	20	200	4

This model can be expressed as a formula:

$$P_c = \frac{a_p \cdot a_e \cdot f_z \cdot v_c \cdot z_c \cdot k_c}{\pi \cdot D_{cap} \cdot 60 \cdot 10^3}$$

where

$$k_c = k_{c1} \cdot h_m^{-m_c} \cdot \left(1 - \frac{\gamma_0}{100}\right)$$

$$h_m = \frac{360 \cdot \sin(\kappa_r) \cdot a_e \cdot f_z}{\pi \cdot D_{cap} \cdot \cos^{-1}\left(1 - \frac{2 \cdot a_e}{D_{cap}}\right)}$$

The following parameters are stored in the SVALI local database as meta-data for a specific milling model:

$$k_{c1} = 1950,\ m_c = 0.25$$

The user installs the validation model expressed as functions as shown in Table 2 applied on the JSON objects *r* received in the stream from the equipment called "mill1":

Table 2. Functions installed in SVALI

Model	**Corresponding function installed in SVALI**
$h_m = \frac{360 \cdot \sin(\kappa_r) a_e \cdot f_z}{\pi \cdot D_{cap} \cdot \cos^{-1}\left(1 - \frac{2 \cdot a_e}{D_{cap}}\right)}$	*Create function hm(Record r) ->Number as 2*pi()*sin(90*pi()/180)*ae(r)*fz(r) / (pi()*dcap(r)* acos(1-2*ae(r)/dcap(r)));*
$k_c = k_{c1} \cdot h_m^{-0.25} \cdot \left(1 - \frac{m_c}{100}\right)$	*create function kc(Record r) ->Number as kc1("mill1")*power(hm(r), -0.25) * (1-mc("mill1")/100);*
measured power consumption	*create function measuredPower(Record r) -> Number as r["power"];*
$P_c = \frac{a_p \cdot a_e \cdot f_z \cdot v_c \cdot z_c \cdot k_c}{\pi \cdot D_{cap} \cdot 60 \cdot 10^3}$	*create function expectedPower(Record r) -> Number as (ap(r)*ae(r)*fz(r)*vc(r)*zt(r)* kc(r))/ (pi() * dcap(r) * 60000);*

The validation function is defined as:

```
create function validatePower(Record r, Number x)
                                -> (Number ts, Number m)
 as select ts, m
    where m = measuredPower(r)
    and abs(x - m) > th("mill1");
```

The function *th(Chartsring k)* is a table of validation thresholds for each kind of machine *k* stored in the local database. After the model is installed in the SVALI server, a CQ validating a single JSON stream delivered from host "h1" on port 1337 is expressed as[1]:

```
select model_n_validate(bagof(input), #'expectedPower',
                                #'validatePower')
from Stream input
where input = corenetJsonWrapper("h1", 1337);
```

Here, the wrapper function *corenetJsonWrapper()* interfaces a data stream server called "Corenet" delivering JSON objects to SVALI.

3.2 Learn-and-Validate

In cases where a mathematical model of the normal behavior is not easily obtained the system provides an alternative validation mechanism to learn the expected behavior by dynamically building a statistical reference model based on sampled normal behavior measured during the first *n* stream elements in a stream. Once the reference model has been learned it is used to validate the rest of the stream. This is called learn-and-validate and is implemented by a stream function with the following signature:

```
learn_n_validate(Bag of Stream s, Function learnfn,
                        Integer n, Function validatefn)
        -> Stream of (Number ts, Object m, Object e)
```

The *learning function*, *learnfn(Vector of Object f)->Object x,* specifies how to collect statistics *x,* the *reference model*, of expected behavior, based on a sequence *f* of the *n* first streams elements.

As for model-and-validate, the *validation function*, *validatefn(Object r, Object x)->(Number ts, Object m)*, returns a pair *(ts, m)* whenever a measured value *m* in *r* is invalid at time *ts* compared to the reference value *x* returned by the learning function.

The function *learn_n_validate()* returns a validation stream of tuples *(ts, m, x)* with time stamp *ts*, measured value *m,* and the expected value *x* according to the reference model learned from the first *n* normally behaving stream elements.

3.2.1 Demonstrator 2

This part demonstrates how learn-and-validate has been applied in an industrial case, based on a cyclical manufacturing scenario. The case was provided by Sandvik Coromant, including the framework, methods, meta-data, needed systems, equipment, and the generated in-process data [2]. The streaming process data used in this demonstrator was simulated in the same way as in Demonstrator 1.

[1] The notation #'fn' specifies the function named 'fn'.

In Figure 3, the blue curve shows the normal behavior of one cycle where the x-axis is time and the y-axis is the measured power consumption. Continuous processing will lead to a certain degree of wear of the equipment. The wear rate is computed by the difference in power consumption between cycles. When the wear rate exceeds an upper limit, indicated by the red curve in the figure, the tool is worn out and should be replaced. Data for this demonstrator was logged using a system from Artis (http://www.artis.de/en/), the visualization in Figure 3 was also generated using that system.

Figure 3. Cyclic behavior curve

The raw cyclic data streams is in this case represented by JSON records *["id":id, "trigger":tr, "time":ts, "value":val]* where *ts* is a time stamp, *id* indicates the identity of a particular machine process, *tr* indicates whether the cycle starts or stops, and *val* is the measured sensor reading to be validated.

Predicate windows: The value *tr* is set by the monitored equipment to 1 when a window starts and 0 when it stops. Such windows with dynamic extents are in SVALI represented as *predicate windows*. Traditional time or count windows cannot be used to identify the cycles when the logic or physical size of the cycle is unknown beforehand and is dependent on a predicate, as in our example. For this SVALI provides a predicate window forming operator *pwindowize(Stream s, Function start, Function stop)->Stream of Window* that creates a stream of windows based on two predicates (Boolean functions) called the *window start condition* and the *window stop condition*.

The window start condition is specified by the *start function*, *startfn(Object s)->Boolean*. It returns true if a stream element *s* indicates that a new cycle is started, in which case *s* is the *start element* of the cycle. The window stop condition is specified by a stop function, *stopfn(Object s, Object r)->Boolean*, that receives the start element *s* and a received stream element *r* and returns true if the received element indicates that the cycle has ended.

For example:

```
create function cycleStart(Record s) -> Boolean
 as s["trigger"] = 1;
create function cycleStop(Record s, Record r) -> Boolean
 as r["trigger"] = 0 and s["trigger"] = 1;
```

In our example, *pwindowize()* is used to build the reference model from the first two cycles of predicate windows. Analogous to the milling example, the CQ validates a JSON stream delivered from host "h2" on port 1338 based on learn-and-validate. It is expressed as:

```
select learn_n_validate(bagof(sw), #'learnCycle', 2,
                                #'validateCycle')
from Stream s, Stream sw
where s= corenetJsonWrapper( "h2", 1338) and
        sw = pwindowize(s, #'cycleStart', #'cycleStop');
```

Learning function: In our example the learning function computes the average power consumption of the *n* first windows *f* in the stream. It has the definition:

```
create function learnCycle(Vector of Window f)
                                -> Vector of Number
as navg(select extractPowerW(w) from Window w where w in f);
```

The function *navg(Bag of Vector)->Vector* returns the average vector of a set of numerical vectors normalized for possibly different lengths. The function *extractPowerW(Window w)->Vector x* extracts a vector of the power consumptions of each element in window *w*. It has the definition:

```
create function extractPowerW(Window w)  -> Vector of Number
as window2vector(w, #'extractPower');
```

The function *extractPower()* is defined as:

```
create function extractPower(Record r)-> Number as r["val"];
```

The system function *window2vector(Window w, Function fe)->Vector f* creates a new vector *f* by applying the function *fe(Object e)->Object* on each element in window *w*.

Validation function: To validate the current stream window, we first extract the power consumption for the current window as a vector and then compare the extracted vector with the learned vector. This is defined as:

```
create function validateCycle(Window w, Vector e)
                        -> (Number ts, Vector of Number m)
 as select timestamp(w), m
    where neuclid(e, m) > th("machine2") and
          m = extractPowerW(w);
```

The function *neuclid(Vector x, Vector y)->Number* returns the Euclidean distance between *x* and *y* normalized for different lengths.

4. PERFORMANCE EXPERIMENTS

To analyze the performance of stream validation in SVALI, the performance of *model_n_validate()* was measured for a set of streams with varying stream rates. Scale-up is simulated by generating many simulated streams with different time offsets based on the raw data provided by Sandvik Coromant. The number of machines is scaled up by increasing the set of streams. The scalability of two queries was investigated:

- Q1: Given the analytical model for the power consumption of a machine process above, produce a validation stream per event of those machines where the power exceeds a threshold 1.2.
- Q2: Given the analytical model for the power consumption of a machine process, produce a validation stream of those machines where the power on average exceeds a threshold 1.2 for 0.1 seconds.

Query Q1 is the example query defined in Sec 3.1.1. Query Q2 uses the following second order functions *measuredPower()*, *expectedPower()* and *validatePower()*:

```
create function measuredPower(Window r)
                    -> Vector of Number m
 as window2vector(r, #'measuredPower');
create function expectedPower(Window r)
                    -> Vector of Number x
as window2vector(r, #'expectedPower');
create function validatePower(Window r, Vector of Number x)
                                -> (Number ts, Vector of Number m)
as select ts(r), m
   where m = measuredPower(r)
   and neuclid(m, x) > th("mill1");
```

Given these three functions, query Q2 validating a bag of streams *bsw* of 0.1 second windows is defined as:

```
model_n_validate(bsw, #'expectedPower', #'validatePower');
```

By simulation, the number of machines is scaled up to 100. Each machine emits a data stream during 30 seconds. To simulate the impact of the performance of streams of different stream rates, the element rate of each stream was randomly chosen between 1 and 10 ms. The validations were done both centrally and in parallel. Central validation first merges streams from all machine processes and then validates them in one process, while parallel validation assigns an independent SVALI process per stream source and then merges the validation streams in a separate process. The parallelization strategy is chosen by the *model_n_validate()* function.

The experiments were made on a Dell NUMA computer PowerEdge R815 featuring 4 CPUs with 16 2.3 GHz cores each. OS: Scientific Linux release 6.2 (Carbon). All simulated stream sources and SVALI nodes run as UNIX processes.

The selectivity of the CQs is defined as the relative stream volume of outgoing tuples from SVALI compared with the incoming ones. Table 3 shows the selectivity of the two queries. The selectivity of the two cases are slightly different because of the randomness in the simulation based on the real data.

Table 3. Query selectivity

	selectivity Q1	**selectivity Q2**
central validation	**14.5%**	**3.4%**
parallel validation	**15%**	**3.5%**

Response time of the validation is measured since low latency is critical because decisions are made when the abnormalities are detected.

For the simple query Q1 Figure 4 shows the average delay (response time) per event caused by SVALI as the number of machines is increased, measured by recording the time when each event arrives to SVALI compared with the time when SVALI emits the corresponding processed event. It shows that Q1 has fast response time but still increases with the number of machines. By contrast parallel validation stays around 0.2 ms as the number of monitored machines increases.

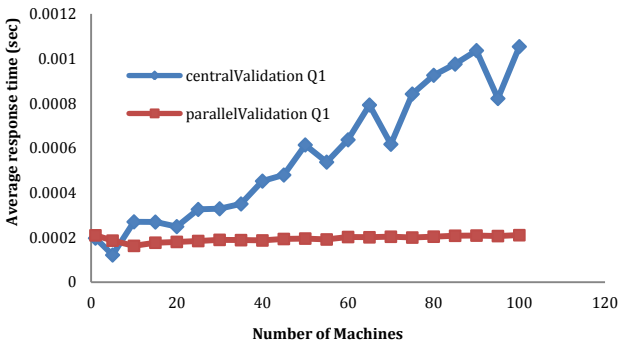

Figure 4. Average response time for Q1

For expensive validations of complex queries like Q2, Figure 5 shows that the central validation does not scale, while the parallel approach remains within bound, i.e. from 1 ms to 2 ms. We also continue increasing the number of simulated machines to explore the capability of our NUMA computer of parallel validation of Q2. In our experiment environment, our system is efficient to handle up to 450 simulated machines.

Both figures show that central validation is slightly faster than the parallel one when the number of machines is small. This is due to the overhead of starting an extra independent validation process for each machine.

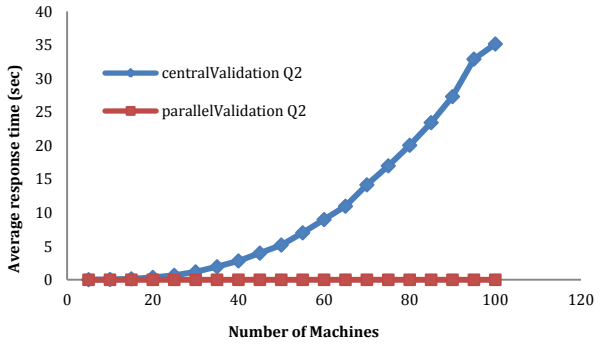

Figure 5. Average response time for Q2

In conclusion, central validation does not scale with the number of machines in particular when validation is expensive, while parallel processing enables scalable validation as long as there are sufficient resources to do the processing.

5. RELATED WORK

This paper complements other work on data stream processing [1] [7] [9] [16] [17] [22] by introducing a general approach to validate normal behavior of streams with non-trivial validation functions.

Several applications of anomaly detection are discussed in [6], such as intrusion detection [8] [11], medical and public health anomaly detection [13] [20] [21], industrial damage detection [12] and so on. Our work belongs to the area of industrial damage detection, i.e. monitoring the behavior of industrial components. Jakubek and Strasser [12] use Neural Networks with ellipsoidal basis functions to monitor a large number of measurements with as few parameters as possible in the automotive field. By contrast, SVALI provides general functionality for monitoring streams from a large number of equipment in parallel, based on plugging in general models.

An adaptive runtime anomaly prediction system called ALERT [19] was developed for large scale hosting infrastructures. The aim was to provide a context aware anomaly prediction model with good prediction accuracy. Rather than anomaly prediction, we mainly focused on supporting online anomaly detection that requires more strict response time. The data streams analyzed in [19] have a fairly low arrival rate, i.e. one sample every 2 seconds and one sample every 10 seconds. By contrast, we show that our system can handle many streams with much higher arrival rates.

Di Wang et al. [20] proposed an active complex event processing system in a hospital environment, where rules are triggered by state changes of the system during CQ processing. In our system, validation models are stored in the SVALI local database and can be modified dynamically by update commands from the application side.

The main focus of [23] is time series data stream aggregate monitoring, while our approach is providing a flexible stream validation framework that can be applied on both individual event monitoring, where only latest event is of interest for processing, and aggregate monitoring, where window aggregation is required for the analysis. This is based on the fact that our stream validation operator treats both raw stream and windowed stream equally.

6. CONCLUSION AND FUTURE WORK

Two general strategies were presented to validate streams from industrial equipment, called *model-and-validate* and *learn-and-validate*, respectively. Model-and-validate is based on explicitly specifying an analytical model of expected behavior, which is compared with actual measured data stream elements. Learn-and-validate dynamically builds a statistical model based on a set of observed correct behavior in streams. We show that the approach is applicable in an industrial setting on real industrial data from real industrial machines.

In our SVALI system, continuous queries validating that equipment behaves correctly are defined declaratively in term of validation functions that are sent to a server, which generates a parallel execution plan to enable scalable computation of validation streams. The experiments show that parallel execution scales better than a central implementation of model-and-validate when increasing the number of streams from monitored machines.

Investigating parallelization of learn-and-validate is future work. Another interesting future work is to regularly refine the learnt model. Furthermore, the impact of complex model functions on the strategy chosen should be investigated, for instance, to

validate streaming data based on trends of measured equipment behavior over time. This can be handled by defining complex model functions that compute trends over time rather than the actual expected measurements. This may involve new scalability challenges.

ACKNOWLEDGMENTS

This work was supported by the Swedish Foundation for Strategic Research, grant RIT08-0041 and by the EU FP7 project Smart Vortex [18].

7. REFERENCES

[1] Abadi, D.J., et al.: Aurora: a new model and architechture for data stream management. The VLDB journal, 12(2), 2003.

[2] Alizadeh, Z.: Method for automated tests of wear (Metod för automatisering av förslitningstest). *Project work in Automated Manufacturing,* 2011.

[3] Alonso-Rasgado, T., Thompson, G. and Elfstrom, B.O.: The design of functional (total care) products. *Journal of Engineering Design*, Vol. 15, No. 6, pp.515-540, 2004.

[4] Alonso-Rasgado, T. and Thompson, G.: A rapid design process for Total Care Product creation. *Journal of Engineering Design*, Vol. 17, No.6, pp.509-531, 2006.

[5] Baines, T.S., Lightfoot, H.W., Evans, S., Neely, A., Greenough, R., Peppard, J., Roy, R., Shehab, E., Braganza, A., Tiwari, A., Alcock, J.R., Angus, J.P., Bastl, M., Cousens, A., Irving, P., Johnson, M., Kingston, J., Lockett, H., Martinez, V., Michele, P., Tranfield, D., Walton, I.M., Wilson, H.: State-of-the-art in product-service systems. Proceedings of the Institution of Mechanical Engineers, Part B, *Journal of Engineering Manufacture*, Vol. 221, pp.1543-1552, 2007.

[6] Chandola, V., Banerjee, A., and Kumar, V.: Anomaly detection: a survey. *ACM Computing Surveys*, 41(3), 2009

[7] Cranor, C., Johnson, T., Spataschek, O., Shkapenyuk, V.: Gigascope: A Stream Database for Network Applications. *Proc. SIGMOD Conf.,* 2003.

[8] Gonzalez, F.A. and Dasgupta, D.: *Anomaly detection using real-valued negative selection*. Genetic Programming and Evolvable Machines 4, 4, pp.383-403, 2003.

[9] Girod, L., Mei, Y., Newton, R., Rost, R., Thiagarajan, Balakrishnan, A.H., Madden, S.: XStream: A Signal-Oriented Data Stream Management System. *ICDE Conf*., 2008.

[10] Helgoson, M., Kalhori, V.: A conceptual model for knowledge integration in process planning, *45th CIRP Conference on Manufacturing Systems*, Procedia CIRP 3 (2012), pp.573-578, Elsevier, 2012.

[11] Hu, W., Liao, Y. and Vemuri, V.R.: Robust anomaly detection using support vector machines. *Proc. of the International Conference on Machine Learning,* pp.282-289, San Francisco, CA, USA, 2003.

[12] Jakubek, S. and Strasser, T.: Fault-diagnosis using neural networks with ellipsoidal basis functions. *American Control Conference*. Vol. 5. pp.3846-3851, 2002.

[13] Lin, J., Keogh, E., Fu, A., and Herle, H.V.: Approximations to magic: Finding unusual medical time series, *Proc. of the 18th IEEE Symposium on Computer-Based Medical Systems*. IEEE, 329-334, Washington, DC, USA, 2005.

[14] Löfstrand, M., Backe, B., Kyösti, P., Lindström, J., Reed, S.: A Model for predicting and monitoring industrial system availability. *Int. J. of Product Development*, Vol. 16, No 2. pp.140-157, 2012.

[15] Meier, H., Roy, R. Seliger, G., Industrial Product-Service Systems-IPS2: *CIRP Annals - Manufacturing Technology*, 59, pp.607-627, 2010.

[16] Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olsten, C., Rosenstein, J., and Varma, R.: Query processing, resource management, and approximation in a data stream management system, *1st Biennial Conference on Innovative Database Research (CIDR)*, Asilomar, CA, 2003.

[17] Shasha, D. and Zhu, Z.: Statstream: statistical monitoring of thousands of data streams in real time, *VLDB Conf.*, pages 358-369, 2002.

[18] *Smart Vortex Project* - http://www.smartvortex.eu/

[19] Tan, T., Gu, X., and Wang, H.: Adaptive system anomaly prediction for large-scale hosting infrastructures. *PODC Conf.*, 2010.

[20] Wang, D., Rundensteiner, E., Ellison, R.: Active Complex Event Processing for Realtime Health Care, *VLDB Conf.,* 3(2): pp.1545-1548, 2010.

[21] Wong, W.K., Moore, A., Cooper, G., and Wagner, M.: Bayesian network anomaly pattern detection for disease outbreaks, *20th International Conference on Machine Learning*, AAAI Press, Menlo Park, California, pp.808-815, 2003.

[22] Zeitler, E. and Risch, T.: Massive scale-out of expensive continuous queries, *Proceedings of the VLDB Endowment*, ISSN 2150-8097, Vol. 4, No. 11, pp.1181-1188, 2011.

[23] Zhu, Y. and Shasha, D.: Efficient elastic burst detection in data streams*, 9th SIGKDD Conf*., 2003.