Flexible On-Board Stream Processing for Automotive Sensor Data

Hendrik Schweppe, Armin Zimmermann *Member, IEEE,* and Daniel Grill

***Abstract*— Vehicle testing and diagnosis requires huge amounts of data to be gathered and analyzed. Not all possibly interesting data can be stored because of the limited memory available in a tested vehicle. On-board preprocessing of data and decisions about which information has to be kept or omitted is thus vital for vehicle testing routines. This paper introduces a method for flexible on-board processing of sensor data of a vehicle. The approach is motivated by sensor network ideas and makes use of stream processing techniques. A processing graph model for automotive applications is proposed, which consists of operator nodes and connecting data streams. This model supplies both recording and processing functionality together. To account for dynamic changes of conditions within a vehicle—most of the time only a small portion of the vehicle states are interesting for diagnosis—both the model and actual software are built in such a way that the whole system can automatically be adapted at runtime whenever certain conditions are detected. The proposed stream processing model has been implemented in a proof-of-concept industrial application, that was deployed to an automotive on-board unit. Results show that this approach effectively trades a little more on-board processing power for a large data volume, that does not need to be saved and transmitted for off-board usage anymore.**

***Index Terms*— On-Board diagnosis, vehicle sensor data, stream processing, data aggregation, embedded systems**

I. Introduction

The complexity of vehicles has increased in recent decades and will continue to increase significantly in the future [1]. Until the late 1960s, cars were basically mechanical systems with a few electrical appliances, e.g., for engine spark and lighting. Modern vehicles are complex electro-mechanical systems with dozens of networked electronic control units (ECUs). ECUs enable or implement vehicle core functions such as power-train control, suspension control, safety, convenience functions, and infotainment. They are connected to a large number of sensors and actuators which they control. ECUs exchange information about their current sensor values over internal networks (for example, a CAN bus), so that multiple redundant sensors are avoided. The types of sensors used in the car are of a great diversity, ranging from pressure sensors over temperature to acceleration and contact sensors.

Data resident and stored on ECUs and data exchanged between ECUs describe, from the technical standpoint, the *state* of the vehicle at any time. To capture, analyze, and interpret this data are important activities of engineering testing and quality control processes. In recent years, the availability of cost-effective in-vehicle and off-board computing and communications systems enabled the systematic acquisition and processing of data from many vehicles over long periods of time. This is particularly important for the late engineering and early production phases of a vehicle's life cycle. The data volume generated from hundreds of sensors, operating at a high frequency (see Fig. 1), is immense. Despite the improvement in computing infrastructure, it is still necessary to utilize filter and data aggregation mechanisms and only record detailed data for specific situations.

Another important application is the area of vehicle diagnosis. Current vehicle diagnosis relies on error codes (DTC, for Diagnostic Trouble Code), that the corresponding ECU sets. These error codes may be retrieved via the on-board diagnostic socket (OBD-II), which became a mandatory standard for the United States in 1996. This diagnostic interface also allows to acquire the various sensor data from the ECUs of a vehicle in real-time.

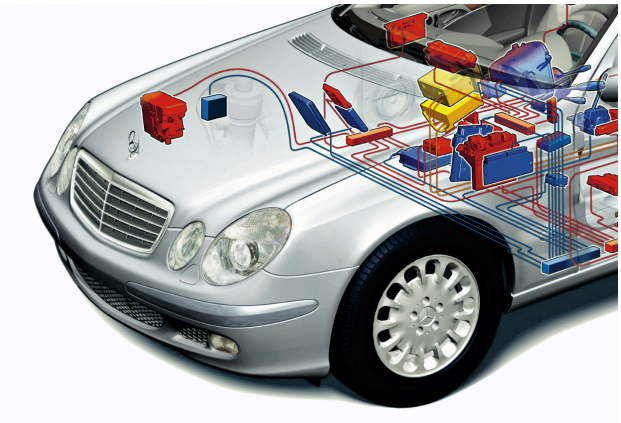


Fig. 1. Bus Systems of W211 vehicle. From DC-Media.

Many approaches for data recording in the vehicle are rather inflexible as far as *how*, *when* and *which* data are gathered, transformed and processed, as they only focus on one of these questions. This work aims at a flexible data aggregation system, that can dynamically adapt its behavior at runtime, depending on the specific state of certain subsystems of the vehicle (e.g. the engine). As a reaction to critical events, our system is able to read and process sensor values at a higher rate and adapt recording. The adaptiveness can also be employed in order to store data selectively.

This work has been performed at Mercedes-Benz Research and Development of North America.

Hendrik Schweppe is with EURECOM in Sophia-Antipolis, France (email: hendrik.schweppe@eurecom.fr)

Armin Zimmerman is with Technische Universität Ilmenau (email: armin.zimmermann@tu-ilmenau.de)

Daniel Grill is with Mercedes-Benz Research and Development of North America Inc. (email: daniel.grill@daimler.com)

Our work aims at solving the industrial problem of vehicle diagnosis—keeping costs low while maintaining flexibility, maintainability and quality of recorded data—by employing and adapting research results from different fields. The main field that caught our interest was stream processing of sensor data. Sensor networks are fundamentally different in their organization, i.e., the network consists of individual, rather autonomous nodes. On the other hand, their functionality, i.e., diagnosing a system state and aggregating raw data wherever possible, is closely related to automotive problems. In our approach we use a single point of access, the vehicle's OBD-II interface, so that we do not interfere with the vehicle's bus infrastructure. Other areas of research that were important for our work are data stream processing and lossy storage of data, which we combined towards a modular and lightweight in-vehicle system.

The main contributions of our work are:

- An *adaptable* stream-based recording system.
- *Situation-dependent* processing and recording of data.
- An event-system that allows *dynamic reconfiguration* on-the-fly.
- *Space-efficient* recording of multidimensional data.
- An *embedded and re-usable* platform, based on the OBD-II/CAN interface and KWP/UDS protocols[1].
- An on-board *prototype* system.

This paper is organized as follows. In the remainder of the introduction, we discuss related work and the state of the art of automotive logging systems. In Section II we introduce the conceptual data-flow architecture and present a formal model of the processing system. Adaptability and dynamic reconfiguration of the system according to specific situations comprise section III. Section IV describes the implementation of the prototype in-vehicle system. An evaluation and selected real-world examples are demonstrated in Section V. We conclude our paper with an outlook on future work. The interested reader is referred to [2] for additional details of our work.

Architecture and Related Work

Existing systems for on-board data recording differ in the quality and quantity of the data they crop. Some systems aim at logging all available data for selected channels to a huge in-car storage, which requires heavy and expensive equipment. For example, an Australian transmission company developed a recording system in 2001, using tape recorders in combination with a Linux PC to record sensor-data at a high resolution [3]. The vehicles were test-driven for up to 100,000 kilometers, during which the tapes with gigabytes of data were regularly sent back to the company for evaluation. A few years later (2004) the same company has, with a different name, put effort into small-footprint logging systems for drive shaft analysis, where data are compressed and approximated prior to recording [4]. As opposed to data-intensive logging systems, there are a number of systems with a small footprint. They record only few data, for example DTCs. General Motors was the first company that introduced such a system: OnStar [5]. They coupled the ability of remote-diagnosis with convenience and security functions for market acceptance reasons. Since 1999, Mercedes-Benz offers a similar emergency and telematics system called TeleAid. The system integrates the vehicle's internal networks with a remote radio interface and demonstrates the significance of software development for safety and convenience systems [6]. Future remote vehicle diagnostic systems will have to provide a more detailed insight into the car's state and might even *"detect the need for preventive maintenance"* [1].

[1]The diagnostic protocols KWP (Keyword Protocol) and UDS (Unified Diagnostic Services) operate on top of the on-board diagnosis interface.

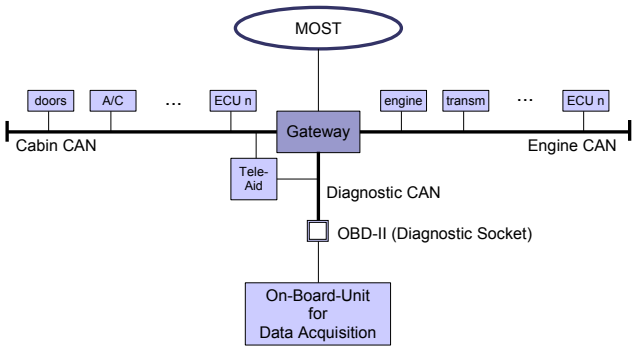


Fig. 2. System Architecture: The On Board Unit (OBU) is connected to the diagnostic CAN bus. It communicates with the ECUs through the Central Gateway.

Bus Architecture

Diagnostic tools are not directly connected to the vehicle's engine or cabin CAN bus system. Instead, they are connected to the OBD socket, which is connected to a separate diagnostic CAN bus. Communication with the ECU is performed over a gateway within the vehicle. Because there is no direct access to the other vehicle buses from the diagnostic interface (i.e., only certain messages are routed), this architecture is considered to be more secure and safe, compared to directly hooking up to, for example, the engine CAN bus. Also, the OBD plug and the diagnostic protocols were standardized in 1996, so that diagnostic devices can be used for different vehicle types and even different manufacturers. While the primary target for the OBD-II standardization were emission control systems, it also covers various other sub-systems. Opposed to that, the CAN communication scheme, called the K-matrix, changes rather frequently. This matrix is needed for decoding data. Because components are often substituted, even cars of the same model have different K-matrices in different years. It would be harder to directly access the CAN buses, because of the mentioned communication decoding and the physical bus access, which can only be made possible with additional costs for normal vehicles. Additionally, one separate connection point per bus would be needed. The generalized bus architecture can be seen in Fig. 2. There are up to six separate CAN buses in today's vehicles.

Diagnostic Protocols

At the time of writing this article, the Keyword Protocol (KWP) is the widely accepted standard for electronic diagnosis

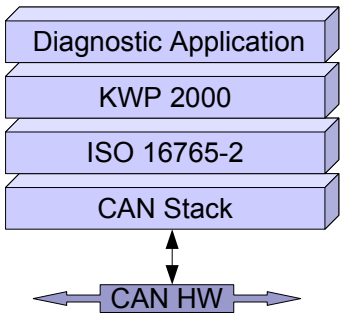


Fig. 3. The Stack of Diagnostic Protocols.

of vehicles [7]. When a diagnosis tool is attached to the OBD socket, it connects to the diagnostic CAN bus and uses the KWP as communication layer. The protocol usually runs on top of the CAN bus, although implementations also exist for other buses, such as LIN or MOST. Communication is performed in a request/response manner. In every request packet, there must be an ECU's identifier, which for a CAN bus is the address ID, and a Service Identifier (SID) (read, write and others). An additional parameter tells the ECU about the precise set of data that is requested. For example the read and write SID require a field called LID (Local Identifier), which maps to memory or registers, whose values are sent back in a KWP response message. Various other services, such as uploading a new firmware to the ECU, are supported by KWP.

Figure 3 shows the stack of protocols used for a diagnostic application. The ISO 15765-2 protocol is used for the segmentation of CAN frames [8]. As a normal CAN frame can only hold 8 bytes of payload and does not support the fragmentation of large data, the glue layer is introduced. An ISO 15765-2 packet can be up to 4 kilobytes big. It divides the data into separate CAN packets with 7 bytes of payload each.

Stream Processing

Stream processing systems [9] define a *computational pattern*. Data are processed on-the-fly, so that the original data, that may be of rather high volume, can be discarded after the computation and only aggregated and filtered data need to be saved or evaluated further. It is related to data flow architectures in a way that data are processed immediately, if available at the inputs.

Requirements and demands for processing streams vary largely by the given constraints and the objectives, so that research in the field of stream processing spreads out rather widely. Golab and Özsu compiled a comprehensive overview of stream processing approaches [9]. On the one hand, there are systems to process large amounts of trading data or position data [10], [11] with real-time quality of service requirements [12] (Aurora, Borealis). On the other hand, stream processing solutions are influenced by traditional relational data base systems [13] (Stanford's STREAM), where so-called continuous queries are applied to data streams. These systems operate on very large volumes of data – for example, think of data streams involving all credit card actions in the US. These systems are designed to process large amounts of data and some support distributed processing of streams. They are not designed for embedded deployment and don't incorporate flexible re-configuration as it is valuable for diagnosis. There exist different approaches in the embedded world. These, however, focus on distributed sensor and processing nodes [14], [15]. One topic of these sensor networks is in-network aggregation [16]. As a matter of fact, the vehicle's infrastructure is a little different. ECUs are not homogeneous by means of the same sensors, processing power and so forth. Also, it is not envisaged or even possible to easily change software on these embedded devices. Especially for diagnosis, a single point of acquisition is defined for diagnostic tools (OBD-II), so that distributed processing is not feasible for our needs.

Large stream processing systems, such as Aurora, Borealis, Stream and TelegraphCQ are considerably too large for an embedded deployment. The most promising work as in-vehicle system is probably VEDAS [17], which follows a distributed data-mining approach for automotive sensor data. Similar to our work, they are exploiting the existing diagnosis infrastructure of the vehicle. In fact, their system also collects aggregated, statistical data, but does not allow an on-the-fly modification of the data processing. Their goal is towards detecting mis-behavior of the vehicle, for example at rental car agencies. This is a rather different focus compared to needs of engineering testing. The Encirq Corporation has designed a distributed automotive data processing system, built around a special query language "DeviceSQL". Their work [18] aims on the distribution of data sources, but does not take *timely behavior* and *on-the-fly reconfiguration* into account. We believe that these are two generally important topics for the automotive domain, especially for engineering testing and diagnosis. We designed our system towards these purposes.

II. A Stream Processing Model for Automotive Sensor Data

The underlying model of our stream processing system is a directed acyclic graph (DAG), which consists of different kinds of processing nodes. These processing nodes *(operators)* are connected by data paths as edges *(streams)*. We call this graph a *Stream Processing Graph (SPG)*. The graph can be adjusted to different situations to allow a varying scope of data processing and recording. Triggers for the adjustment of the graph are defined as part of the graph itself (see below).

The extension of the stream processing graph by event-action pairs allows an adaptation of the graph at runtime. We introduce event triggers for the re-arrangement and re-configuration of the SPG. The two major problems of previous recording systems in the vehicle are addressed: *Customized* aggregation and filtering of data is now possible and actions may be taken upon *individually defined* events.

We will first present a formal model of the graph. The global system design has been carried out in a modularized manner, with protected queues as comunication channels, as pointed out in [19]. In detail, we follow a data-flow oriented approach, which is similar to, but weaker than the SPF (Stream Processing Function) algebra of Broy et al., a so-called Basic Network Algebra as discussed in [20]. They

specify a complete calculus towards concurrent computation. We don't need such a fine granular model and do not adopt, for example, direct data feedback loops that explicitly are part of the SPF algebra. We use control inputs and an adaptation of the graph's topology and operator node's parameters as feedback principle. Modifications of the graph are described in Section III. Our model and the terms used throughout the paper are inspired and adopted from operational stream processing research approaches, that have been presented in the previous section. In detail, these are [9], [10], [11], [12], [13] and [21], where the latter gives a good introduction to the stream processing paradigm. In contrast to these approaches, our model allows for dynamic reconfiguration of the processing structures at runtime.

Definition: A *Stream Processing Graph* is a DAG $G = (N, E)$ consisting of nodes N and edges E. Edges are a subset of all possible connections between the nodes $E \subset N \times N$.[2]

Nodes N of the graph represent *operators*, which process *data streams*. Edges of the graph represent the flow of data between operators. An edge $e = (n_1, n_2)$ represents the flow of data between node n_1 and node n_2. We denote the space of all possible graphs by $\mathbb{G}$.

Nodes without incoming edges are *source nodes* and nodes with no outgoing edges are *sink nodes*. We say that a node with in- and outputs is an *inner node*. There needs to be at least one source and one sink node within an SPG. As a matter of fact, data flows from source nodes to sink nodes. Source nodes produce data obtained from sensors or alike, while sink nodes take *action* on arriving data. The most common action is to *save* incoming data. Inner nodes of the graph process data from their incoming edges and output processed data to outbound edges.

Data within an SPG are encapsulated as data packets, called *tuples* $\mathfrak{t}$. They are called tuples, because they do not only contain the data but also meta data, such as a time-stamp of creation.

Definition: A *tuple* $\mathfrak{t} = (val, \tau, m) \in T$ consists of a value val, a time-stamp τ and a mileage stamp m, which are addressed with $\mathfrak{t}.val, \mathfrak{t}.\tau$ and $\mathfrak{t}.m$. The set T is the tuple space.

Definition: A *stream* $s = (\mathfrak{t}^*)$ consists of an ordered sequence of tuples $\mathfrak{t}^*$, so that for all tuples $\mathfrak{t}_1, \mathfrak{t}_2 \in s$ that fulfill $\mathfrak{t}_1 \leq \mathfrak{t}_2$, the tuple $\mathfrak{t}_1$ will be before $\mathfrak{t}_2$ in s. The relation $\leq$ reflects both time and mileage, i.e. $\mathfrak{t}_1.\tau \leq \mathfrak{t}_2.\tau$ iff $\mathfrak{t}_1.m \leq \mathfrak{t}_2.m$. S denotes the set of all streams. At any given time, the bijective function $m_e : E \mapsto S$ maps all edges of the graph structure to corresponding streams. The sets of input and output streams of a node n are defined as:

$$S_n^{\text{in}} = \{s \in S | \exists e = (\cdot, n) \in E : m_e(e) = s\}$$
$$S_n^{\text{out}} = \{s \in S | \exists e = (n, \cdot) \in E : m_e(e) = s\}$$

Definition: An *operator* o associated with a node n is given by $o = (f_n^{\text{select}}, f_n^{\text{output}}, f_n^{\text{update}})$ at any time t. The according function spaces are denoted as F^{select}, F^{output} and F^{update} respectively. Operators have an internal state σ, which is introduced below. To *execute* an operator means to update output streams and the inner state of the node as follows.

Generating a tuple, i.e., adding it to an output stream s, triggers the function f_n^{select} for the succeeding nodes n, for all $n : s \in S_n^{\text{in}}$. This function decides whether the operator is executed. It maps the input streams of the node to a modified set of streams, which consists of the tuples selected to be processed by the operator.

$$f_n^{\text{select}}(S_n^{\text{in}}) = S_n^{\text{in}^\circ}$$

Only if data is available from any input streams, i.e. at least one of the member-sets of S^{in° is not empty, the operator itself is executed. The tuples are taken out of the input streams: $\forall s_i \in S_n^{\text{in}} : s_i' = s_i \setminus s_i^\circ$, where s_i° denotes the corresponding tuple-set from $S_n^{\text{in}^\circ}$.

Operators may be configured by parameters $p \in P$. For example, a predicate for filtering the stream's data or a window for aggregating data from within that specific window can be given as a parameter.

We distinguish between *stateful* and *stateless* operators. A state is, for example, a preliminary output result. The state $\sigma_n \in \Sigma$ of a node n is changed to σ_n' by the update function

$$f_n^{\text{update}} : S^{|S_n^{\text{in}}|} \times \Sigma \times P \mapsto \Sigma$$

The output of an operator is generated by its function

$$f_n^{\text{output}} : S^{|S_n^{\text{in}}|} \times \Sigma \times P \mapsto S$$

The output of an operator is usually only a single tuple. The operator node's output streams s changes to s' so that

$$\forall s \in S_n^{\text{out}} : s' = s \cup f_n^{\text{output}}(S_n^{\text{in}^\circ}, \sigma', p)$$

This means $m_e(e)$ now maps to the stream s' for the graph's corresponding edge e. All output streams are appended with the same tuple.

Similar to the function $m_e(e)$, there exist functions

$$m_n^{\text{select}} : N \mapsto F^{\text{select}}$$
$$m_n^{\text{update}} : N \mapsto F^{\text{update}}$$
$$m_n^{\text{output}} : N \mapsto F^{\text{output}}$$

These functions map the according semantic to the operator node at a given time.

We allow a dynamic reconfiguration of the graph. The semantic of reconfiguration depends on updating the state of affected operators accordingly. A reconfiguration therefore will always update the operator's state for stateful operators. For every action and every operator a parameter-state-update function $f_{\text{p-s-upd}} : P \times \Sigma \times P \mapsto \Sigma$ is needed. It updates the old state, that originated from the old parameters, to a new state, so that the operator's consistency is maintained. For example, when changing the window size of an operator, the internal state has to be modified in a way that either the preliminary result is modified or the state is reset to the initial value.

The operators assigned to source nodes supply the vehicle sensor data. One can think of their states as externally controlled (i.e., by the environment and state of the vehicle).

[2]It is a real subset, because a complete graph is not a valid graph structure within the posted limitations.

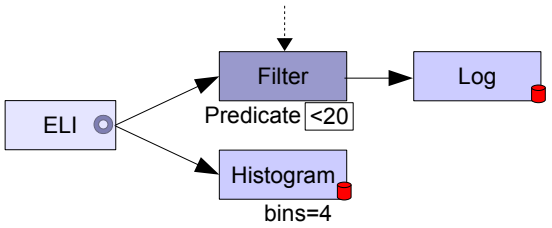


Fig. 4. Example 1: Sensor data (ELI: Engine Load Indicator as percentage) are sent to a filter as well as to a histogram. Only tuples that match the predicate are logged (in this case: low-load state), while the distribution of total engine load is maintained in the four-bin histogram. The filter's predicate may be adjusted through the control input depticted above. It is used by a different end-operator (not shown).

They execute whenever new data were gathered (i.e., according to a sensor's sampling interval). The nodes on the other end of the graph, the sink-nodes, are end-operators. They take action on incoming tuples, which is the core of the graph's reconfiguration strategy.

Example of an Operator: Assume a filter predicate is $p_{\text{filter}} = \boxed{< 20}$. The update function $f^{\text{update}}_{\text{filter}}(\mathfrak{t}, \sigma, p)$ has no influence on the operator's behavior (as it is stateless). The output function is defined as:

$$f^{\text{output}}_{\text{filter}} = \{\mathfrak{t} \mid \mathfrak{t}.val \boxed{< 20}\}$$

This means that the operator produces an output iff the predicate is true. Another example, where an internal state is maintained by the operator, is presented below.

Graphical Representation of SPGs: We have adopted the graphical representation of operators and streams as *boxes and arrows* that was suggested in [21], [22] and [11]. The data flows from source nodes (operator boxes on the left) to sink nodes on the right. We introduce another kind of arrow: We allow to change the node's behavior via a *control input*, that is depicted as a dashed arrow at the affected operator(s). The so called *actions* that make use of these control inputs are explained in III.

A small SPG is shown in Fig. 4 as Example 1. The input sensor source ELI (Engine Load Indicator, a percentage value internally calculated by the engine ECU) is branched to two operators: A filter operator and a histogram. There is a simple logging store connected to the filter operator. The filter will only output tuples that fulfill its predicate $\boxed{< 20}$. The filter's predicate can be changed by the control input denoted above. It can be changed upon an event condition that is not part of the figure. The histogram and log store do not have outgoing edges (they are sink nodes) and record incoming data.

Window Operators

Operations on data streams in automotive applications are typically performed on so-called *windows*. This allows, for example, to reduce a sequence of tuples from the input to only a single tuple as the output. Every operator with a given window size > 1 holds a state σ, which represents either the current preliminary result or buffered, unprocessed input tuples for later batch processing, e.g. when the calculation requires more than one pass over the data. The state is updated whenever the operator function processes incoming tuples or produces outgoing tuples.

A window consists of a finite number of consecutive tuples of a stream. The tuples of a stream are assigned to windows upon arrival. They are processed with respect to these windows. We denote the *size*[3] of a *window* by ω.

Windows can be defined over different attributes. They are typically defined over the *time* dimension: the window size is given as an interval size in time. However, a window size can also be defined by a *count*. Then the window of size ω consists of exactly ω tuples. In that case, the time difference of the tuples within this count-window is variable as opposed to the number of tuples, which is fixed. A third dimension, most important in the automotive application context, is *mileage*. For determining whether a tuple is within the boundaries of a specific window, their time or mileage stamp is compared to the window's start time or mileage. We call the dimension, over which a window is defined, the *windowing attribute* D.

The time stamps of tuples inside a window are less than or equal to ω apart, so that for the first tuple $\mathfrak{t}_f$ and the last tuple $\mathfrak{t}_l$ of a time-based window, the expression $\mathfrak{t}_l.\tau - \mathfrak{t}_f.\tau \leq \omega$ is always valid[4].

An incoming tuple $\mathfrak{t}$ falls into a window w if the tuple's time stamp $\mathfrak{t}.\tau \in [t_f.\tau, t_f.\tau + \omega]$, where $\mathfrak{t}_f$ is the first tuple of window w. If the tuple does not fall into window w, then w is *closed*, because the stream's metadata are monotonically increasing. This means that the window size is reached, i.e., the distance of the time stamps of the first ($\mathfrak{t}_f$) and the new tuple ($\mathfrak{t}$) of w is at least ω: $\mathfrak{t}.\tau - \mathfrak{t}_f.\tau > \omega$. The window can now be processed by the operator's aggregate function, which may require more than one pass over the data. If only one pass is required, an iterative calculation of the result may be performed, so that the tuples of the current window do not need to be saved[5]. The operator's function f calculates a new tuple, which is the output of the operator – for example, the mean value within the given window.

We allow more than one window to be open at a time, resulting in overlapping windows. The parameter *window slide* δ indicates when (i.e., after how much time elapsed since the start of the last window) a new window shall be opened. For the simple case $\delta = \omega$, windows are non-overlapping, so that there will be exactly one open window for processing at a time. These windows are called *tumbling windows* [12] [23].

Example of a Stateful Operator: Now that we have introduced windows, we will give an example of how they can be used within stateful operators. We define an average-operator AVG that takes the average of an input of $\omega = n$ tuples as follows. The according parameters are $\omega = n$, $D = count$, $\delta = \omega$. For this operator, the internal state consists of two parts,

[3]The window size is sometimes also called a window range [23].

[4]To simplify matters, we use the windowing attribute *time* in our examples. Our considerations also apply for the other windowing attributes.

[5]Even when the result can be calculated iteratively and an additional buffering within the operator is not needed, it can still be advantageous, because it allows to change the window size in operation without a loss of data, as discussed in III-B.

$\sigma.sum$ and $\sigma.cnt$. For the initial state σ_{init}, both values are zero. AVG has one input stream, for which f^{select} did produce $\{\{\mathfrak{t}\}\}$ as output.

$$f_{\text{AVG}}^{\text{update}} \quad (\{\{\mathfrak{t}\}\}, \sigma, p)$$

$$= \begin{cases} \sigma'.sum = \sigma.sum + \mathfrak{t}.val \\ \sigma'.cnt = \sigma.cnt + 1 \end{cases} \quad \text{if} \quad \sigma.cnt \leq n$$

$$= \begin{cases} \sigma'.sum = \mathfrak{t}.val \\ \sigma'.cnt = 1 \end{cases} \quad \text{if} \quad \sigma.cnt > n$$

$$f_{\text{AVG}}^{\text{output}} \quad (\{\{\mathfrak{t}\}\}, \sigma', p)$$

$$= \begin{cases} \{\} & \text{if} \quad \sigma'.cnt \neq n \\ \{(\frac{\sigma'.sum}{\sigma'.cnt}, t_{now}, m_{now})\} & \text{if} \quad \sigma'.cnt = n \end{cases}$$

III. Dynamic Reconfiguration

We now describe how our concepts of graph adaptation and recording of data are modeled. Sink nodes (without outgoing edges) are so-called *end operators*. The first possible function of an end operator is to simply store the incoming tuples. However, in our approach they are also used to trigger changes of the stream processing graph during run time. This can be done by either updating numerical parameters of other operator nodes, or by altering the graph structure in itself.

At the end operators of an SPG, data has already been filtered and processed and thus *"each path from a sensor input to an output can be viewed as computing the condition part of a complex trigger."* [22]. End operators represent event-condition-action (ECA) rules. The arrival of a tuple on the input stream may be regarded as a *basic event*. One event triggers one or more actions. In order to be more flexible, the actions taken upon a "tuple-arrival-event" may also depend on boolean conditions c_i.

Definition: An *end operator* o_e is defined by at least one input stream and a set of condition-action rules (c_i, a_i). A condition c_i is a predicate defined on tuples of the input stream. If $c_i(\mathfrak{t})$[6] is true, action a_i of the pair (c_i, a_i) is executed.

In case of rules $r_i = (c, a_i)$ and $r_j = (c', a_j)$ with $c = c'$, both actions a_i and a_j are performed. This is abbreviated as $r = (c, \{a_i, a_j\})$. Action a_i of the rule $(\top, a_i)$ is executed unconditionally. This means that the basic event, the arrival of any tuple, causes the execution of the action in this case.

We distinguish three types of actions, that may be taken at an end operator:

- *Storage* actions $a^{\mathcal{S}}$, which do not modify the processing pattern of the graph,
- *Parameter modification* actions a^P and
- *Topology modification* actions a^T,

of which the latter two change the system's behavior.

[6]For n input streams it would be $c_i(\mathfrak{t}_1, \ldots, \mathfrak{t}_n)$ with tuples from the different streams.

A. Storage Actions

Storage actions $a^{\mathcal{S}}$ conform to the signature $a^{\mathcal{S}}(\mathfrak{t}, \mathcal{S})$, where $\mathcal{S}$ is a store location (e.g. memory or a file). The content of the store is addressed by $[\mathcal{S}]$. A simple storage operator, as shown in Example 1 in Fig. 4, only *saves* all input data. It consists of an empty condition (i.e., $c = \top$) and an action to save the input tuple. Saving is done by appending data to the store. The store $\mathcal{S}$ can be regarded as logfile, which $a^{\mathcal{S}}(\mathfrak{t}, \mathcal{S}) = ([\mathcal{S}]\ concat\ \mathfrak{t})$ appends to. Another kind of storage operator, that was used in Example 1, is an equi-width histogram. It is adaptable by the number of quantization levels, a minimum, and maximum value.

B. Parameter Modification Actions

An action $a^P(N, P)$ with $P = \{(p_1, \ldots, p_n)\}$ modifies the parameters p_i for every operator node $n \in N$.

If the parameters of a stateful operator are changed, its state σ has to be modified, too. It depends on the type of operator state (incremental or buffering), whether for example decreasing the window size ω is possible consistently. If the operator holds all previous tuples of all open windows in its state, a batch processing of the old tuples with the new window parameters can be performed. If the state of the operator is only a partial result of its computation, then it is not possible to decrease the window size consistently. There will be a gap in calculation to the next subsequent window result. Increasing the window size and changing the parameters of stateless operators (e.g., filter predicate, map function) are always possible. The afore mentioned function $f_{\text{p-s-upd}}$ updates the state of the operator. It is given for every operator that is affected by the parameter modification action. The new state is calculated from the old state and the old parameter set.

C. Topology Modification Actions

Actions a^T map $\mathbb{G} \mapsto \mathbb{G}$. The graph structure of the SPG is modified in a way that either new nodes and edges are inserted and connected, or specific nodes or edges are deleted. The graph requires also an adequate modification, for example when an inner node is deleted, by inserting and deleting edges. Actions a^T must always modify the SPG in a way that the graph stays consistent (i.e., inner nodes have a sufficient number of inbound edges and at least one outbound edge).

Graph modifications may either add or delete a node n_{mod}, as explained below.

Inserting a node: To add a node n_{mod} to the SPG, a sorted set of preceding nodes $N^- \subseteq N, |N^-| = k$, where k is the necessary number of inputs for the operator, has to be supplied by the action. Additionally, the set $N^+ \subseteq N$ has to be specified, which contains the operator nodes that shall receive the new operator's output. The resulting graph $G' = (N', E')$ is:

$$\begin{aligned} N' &= N \cup \{n_{\text{mod}}\} \\ E' &= E \cup \{(n_i, n_{\text{mod}}) | n_i \in N^-\} \cup \{(n_{\text{mod}}, n_o) | n_o \in N^+\} \end{aligned}$$

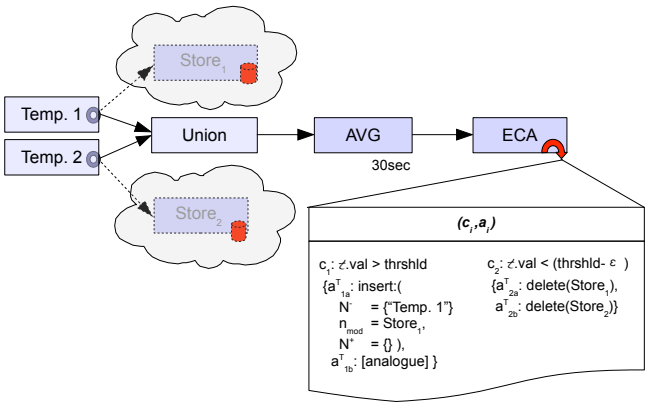


Fig. 5. Example 2: Topology Modification. Upon the conditions c_1 the additional storage nodes Store_1 and Store_2 are created. Condition c_2 is the counter-condition, that removes the storage nodes.

Deleting a node: Deleting a node n_{mod} modifies the graph $G = (N, E)$ in the following way.

$$\begin{aligned} N' &= N \setminus \{n_{\text{mod}}\} \\ E' &= E \setminus \{e = (n_1, n_2) | n_1 = n_{\text{mod}} \vee n_2 = n_{\text{mod}}\} \cup E'' \end{aligned}$$

The node n_{mod} is removed from the set of nodes and all incoming and outgoing edges are removed from E. The set of new edges E'' is to fulfill the consistency of the SPG. It may be empty, for example, if node n_{mod} had no successors (i.e., n_{mod} was a sink node).

Example 2: An intuitive example for modifying the graph is *adding* and *deleting* additional *storage nodes* for recording the streams at any source or inner node of the SPG upon condition and counter-condition. The graphical representation of this graph can be found in Fig. 5. By this selective adding of additional recording points, specific situations, such as "at a high temperature", may be monitored selectively and in detail. Two temperature values are merged into one stream in pairs by a union operator. This results in a stream that carries data from both streams (e.g., left and right cylinder head temperature). The adjacent operator averages the incoming tuples over a 30-second time-window. Now if one of the resulting tuples exceeds a threshold and $c_1 : \mathrm{t}.val > thrshld$ comes true, the actions a_{1a}^T and a_{1b}^T are taken. These actions each add a new node $n_{\text{mod}} = \text{Store}_i$ to the first and second temperature sensor, so that an individual monitoring of the two sensors for this situation is achieved. An opposite condition is shown as c_2: If the value drops under $thrshld - \epsilon$, the graph is restored by deleting the storage nodes[7].

IV. Implementation

We have implemented a prototype system on an embedded Linux telematic device, which is equipped with a CAN transceiver. It is equipped with a 150MHz TriCore processor,

[7]The ϵ is considered for hysteresis. Otherwise a bouncing between the two conditions would possibly occur and constantly insert and delete the storage operator nodes.

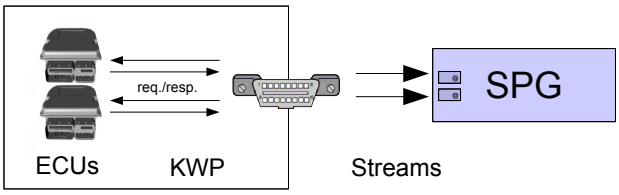


Fig. 6. Acquisition Module: The request/response polling of data from the ECUs via KWP is decoupled from the actual stream processing engine.

128MB of RAM and a CF slot for data storage. Via USB it can be enhanced by additional hardware, e.g., an IEEE 802.11 network interface. We connect to the OBD-II's diagnostic CAN bus and use the diagnostic protocols KWP and UDS to query the various ECUs. The ECUs are periodically queried for their sensor values, as schematically shown in Fig. 6. We separated these two parts in system design to allow re-simulation with specific rides where all data have been recorded. These values comprise the input of the processing graph, i.e., the source nodes of the SPG. The total volume of sensor data that can be acquired by the system depends on a) the vehicle's bus topology, b) the load of the CAN buses, and c) the processing load of the individual ECUs, which may reject requests for sensor data when they are busy. Because of the arbitration mechanism of the CAN bus, security and safety relevant data are always prioritized before our requests. In average, we obtain data with a frequency of 1 Hz. This is of course not sufficient for an in-depth analysis of, for example, faulty lambda sensor loops. It is, however, very adequate for detecting most faults and possible relations between faults – especially considering that no modifications of the vehicle have to be made.

The programming language C++ was used for the prototype. Inheritance allowed an encapsulation of communication and control flow structure of the SPG and thereby separating it from the operator functionality. All operators inherit functions that allow to connect them to any other operator of the SPG and safely operate on streamed data tuples. A "window operator" class, for example, encapsulates operations on sequences of tuples, such as keeping track of currently open and to-be-processed windows. Custom operators inherit these functions and only need to implement code for processing the input data accordingly. In Fig. 7 we show a UML class diagram, which explains how inheritance is used in our implementation. We show an exemplary set of classes and, for two of them, their corresponding attributes and function. The operator class *Op* provides all functions to interconnect the stream processing graph. Other classes enhance different groups of operators (e.g., the storage operators) with a common interface. The *Command* class implements actions on the SPG itself and, for certain actions, on specific families of operators, e.g., with buffered storage operators to save their state.

For concurrency, we use POSIX pthreads and follow a different approach for the execution for Linux kernel series 2.4 and 2.6. The Linux 2.4 kernel series, which was used on the on board unit, does not support lightweight threads [24]. Due to this, we used threads conservatively with respect

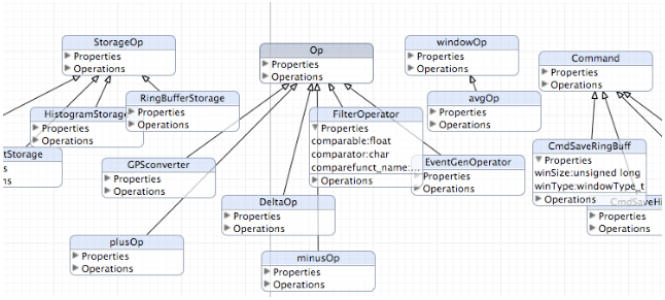
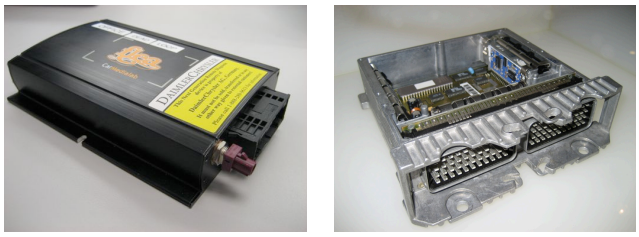


Fig. 7. UML class diagram extract. The concept of multiple inheritance was used, so that e.g., individual storage operators inherit graph functionality from *Op* and a storage interface from *StorageOp*. Actions are taken via the *Command* interface.

to the embedded environment on v2.4. We use threads only for the sensor source operators, which trigger the execution of subsequent operators throughout the SPG. For v2.6, we employ threads for every operator, so that operator can be scheduled individually without relying on sensor input to trigger the execution. Apart from that, the event-system is always scheduled separately, so that a timely execution of actions is ensured. While an event changes the processing of the graph, for example, adjusts parameters and state of an operator, the specific operator is held from being executed and thus maintains a deterministic state.



(a) On-Board Unit (b) Engine ECU

Fig. 8. The On-Board Unit acquires data from the various ECUs within the vehicle via KWP and UDS.

Actions that change the stream processing graph's topology or parameters may be taken on event conditions, which have been defined as a path within the SPG. The afore mentioned "basic event" of a stream triggers actions. Every event is mapped to one or more actions, which themselves affect one or more operators of the graph. To give an example, an action to save time-series is applied to a set of ring buffers (default: all) for a defined window (default: 30 seconds). The same event-action rule can be triggered from different locations within the SPG (e.g., 'drop of tire pressure' or 'high lateral acceleration').

We implemented a number of processing operators, e.g., a filter operator with customizable predicate, a delta/gradient operator and window-aware versions of the standard functions *MIN, MAX* and *AVG*, along with a number of recording operators.

For complex technical problems such as engine management, it is often necessary to look at a *combination* of different sensor values (e.g., for engine RPM, throttle position, and intake manifold pressure). We implemented a *multi-dimensional histogram* storage operator, that allows multiple input streams with individual quantization attributes $(min,\ max,\ num_{bins})_o$. The input streams are synchronized within time windows. Because a multi-dimensional histogram is a very sparsely populated data structure, we had to store the data in an intelligent way, that a) allowed to save data space-efficiently, i.e., only to store non-zero elements and b) has an acceptable access time, i.e., does not depend on the size of the histogram. We used a trie [25], a search tree, to store the histogram. The index of the histogram is stored as key at the edges of the tree. One byte is used as quantization index of each input. The actual count of a histogram bin is saved in the leaf nodes of the tree. The access time of the trie only depends on the depth, i.e., the number of input streams, which makes it suitable for recording long time intervals.

V. Experiments and Evaluation

We have conducted a number of experiments on the road, which showed that our approach is suitable for different scenarios of data collection. In the following, we will give an example, of how to apply our SPG model to a given problem. The data that we present as part of the evaluation is deliberately modified, so that confidentiality and proprietary rights are preserved. After the use-case study, we show a quantitative analysis by means of comparing recording techniques, i.e., the SPG-based approach, selective recording on the diagnostic application layer and raw data recording with respect to data volume to be transmitted. We show how worst-case response time of ECUs compares to the observed latency, and how the size of an SPG affects memory consumption.

The problem of qualifying intermittent and non-reproducible errors is inherent to the automotive industry. We have taken up cylinder misfires as an example for indeterministic events, that we want to investigate. Knocking and misfires, against the common perception, are still quite common for modern engines, as the combustion is usually most efficient when it happens close to the knocking limit [26].

Objective: To obtain a profile of the situation, where a specific error (i.e., misfire in our example) occurs. The profile must be informative enough to find a lead to the root-cause of the error, while its data volume must not exceed the available storage capacity.

Rationale: A complete log of the vehicle's environment data is not sensible. Actually, even short time-series for every occurrence will produce large quantities of data for a high event-frequency, for many sources, and over a long time-span.

Approach: We use a multi-dimensional histogram as state-space, that represents a partial state of the vehicle. The non-zero states represent those in which an error situation has occurred. Ring buffers provide the input to every dimension of the state space. A ring buffer may be placed anywhere within the SPG, so that sources (e.g., speed) or pre-processed streams (e.g., average or maximum of speed within the last minute) can be used as input. For every dimension of the state-space, there are parameters for the minimal and maximal value and the number of quantization levels. One may, for example, only be interested in a rough quantization of the engine temperature

(cold, warm, hot), but need a finer granularity for other dimensions. By using these additional a-priori information about the number of necessary discrete states, much storage space can be saved.

Implementation: The engine ECU supplies a misfire-counter for every cylinder. We monitor the counters for changes and—upon an increment of a counter—we trigger an update of the state-space via an event/action pair.

Fig. 9. The event 'MISFIRE' is triggered, if misfire counter (MFC) of cylinder i was incremented. Not shown: The according action triggers an update of the state-space with data from various buffered sensor sources.

Four consecutive operators are used per cylinder, to set off the event (Fig. 9). The SPG-arrangement is started by the source operator that supplies the misfire counter value, a delta, a filter and an event operator. Tuples periodically arrive at the delta operator, which outputs the difference between two following tuples to the filter. They are dropped, if their value is zero. Otherwise—this indicates the misfire—the tuple is forwarded to the event operator, which propagates the event "misfire" to the event handler. The event handler performs an action that releases the most recent data tuple from ring buffers via their control input, which are themselves the input to the multi-dimensional histogram store.

Results and Interpretation: The evaluation of a multi-dimensional state-space reveals patterns, on *how* the vehicle was used when errors occurred. A clustering of occurences within the state space points out potential root-causes of the problem.

We used a set of seven sensor sources for the state space. In Fig. 10, we show a plot of two dimensions (a projection on this plane) of the error profile recorded for misfires. The profile shows a cluster of misfires around 3400 rpm for all speeds. The second cluster at low engine revolutions does not spread over all vehicle speeds, and concentrates on the low (0-30 km/h) speeds.

As a subsequent question, one may ask *why* the misfires do occur mostly in the interval around 3400 rpm. An obvious guess for reasons of misfire is that they occur at excessive

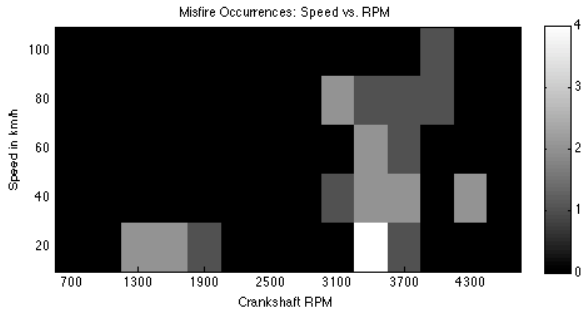


Fig. 10. Two-dimensional histogram profile for sensor readings of the vehicle's velocity and crankshaft RPM, sampled, whenever a misfire occurred: Every misfire is represented by a non-zero, i.e., non-black field (displayed as grey scale value). Misfires seem to occur more frequently at around 3400 RPM.

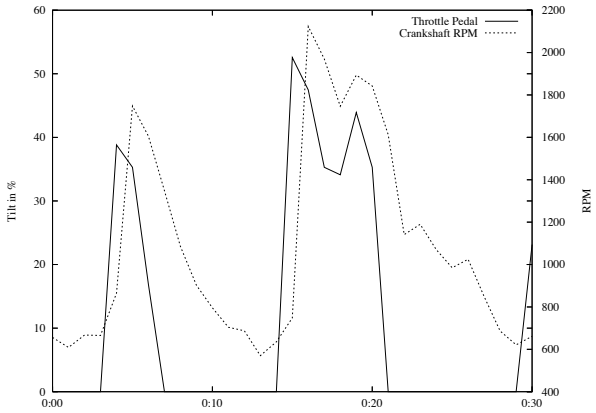


Fig. 11. Engine RPM (right scale) and Throttle Pedal tilt in percent (left scale), thirty seconds before a misfire occurs. The steep and high peaks indicate abrasive driving.

changes of engine revolutions or load of the engine. To prove this guess right, we added an additional action to the misfire event, which records a *time-series* from a buffered sensor-source (see Fig. 11 for a plot of engine RPM and accelerator pedal percentage). We can see that before the misfire occurred, a rapid change of the engine speed has happened. One may presume that the engine load also changed rapidly, during these thirty seconds before the misfire.

As far as a reduction of collected data is concerned, it may even be adequate not to record time-series, but only to record the maximal gradient for a recent time interval. The SPG-design easily allows to insert a delta operator with specified time-window (e.g., two seconds) and a maximum operator—with window size equal to the ring-buffer's—just before the final log operator. This way, only one value (i.e., the maximal gradient) would be saved instead of a complete time series.

Performance Aspects:

Let us consider the data that needed to be recorded for the preceding example. We have observed a number of 19 misfires during a period of one day. To record the two-dimensional state-space (using a naïve implementation—it is even less with the trie structure we used), one needs $14 \times 5 \times 2 = 140$ bytes (this corresponds to a resolution of 14 levels for RPM and 5 for the speed and a two byte value as counter). Plus, for every occurrence of a misfire, a 30 second time series of two values (crankshaft RPM and throttle pedal tilt, both as two byte value and with 1 Hz resolution) is recorded. This makes an additional $19 \times 30 \times 2 \times 2 = 2.280$ bytes for the 19 misfires. Compared to that, a raw recording of these selected values would need $3.600 \times 8 \times 2 = 57.600$ bytes per hour[8]. A complete recording of the power train CAN bus data would result in

[8]Each cylinder's MFC produces 2 bytes per second and we assume a 6 cylinder engine.

SPG	Sel. KWP LIDs	CAN complete
140 + 120 / event	57600 / hour	75.825.000 / hour

Fig. 12. Data volume (bytes) to be saved and transferred for the presented experiment. Use case: Profiling of misfires with two data sources to be recorded when the misfire event takes place. Please keep in mind that for the SPG use case, event-based recording is used, where with KWP the input sources have to be recorded continuously. The same is true for the third column, where no filter is applied during recording.

around 76MB[9] per hour.

The maximal data rate which can be achieved when using diagnostic protocols for the acquisition of sensor data depends on multiple factors. An ECU itself is required to answer every diagnosis packet within a response time of $RT = 20ms$ [8]. The response packet may also deny the request (such as "busy, answer is delayed"). More influential, however, are the network topology and the size of request's response messages. In today's vehicles, there exist multiple gateways between the diagnosis interface and an ECU. Each of them will add a maximum of $5ms$ of latency. The allowed roundtrip time for an ECU behind two gateways therefore is $RTT = RT + 2 \times 5ms = 40ms$ (neglecting bus load and possible interference). For typical "read local identifier" requests that carry a number of signals, a request is split into multiple consecutive network frames, which is in a way similar to TCP windows. Between these consecutive frames, a suggested separation time of $ST = 16ms$ has to be kept. A frame of 40 bytes needs to be split into 6 frames (there are seven bytes of payload, as defined in [8]. One byte is used for KWP control information, such as frame counter. The first frame only has 6 bytes of payload.) The first frame needs to be acknowledged, so this adds an additional round-trip time and we have a total of $2 \times RTT + 4 \times ST + RT = 164ms$. This means that the frequency to be achieved in this case is 6 Hz. Our experiments have shown, that this rate is more a theoretical one. In fact, we found the response time for a single packet request to be normally distributed around 7ms—despite two gateways. However, diagnosis traffic may for example be delayed by higher priority bus load or simply by the ECU's flow control, as in KWP an ECU has the freedom to send a negative response code at any time. We observed, for an ECU behind two gateways and with 40 byte response packets, an average response time of 379ms, which corresponds to a recording rate of 2.639 Hz.

The memory footprint of our system—including an XML parser and without optimizations—is 2972 bytes. For every operator that is added to the graph, around 15kb of additional memory are needed. The actual size depends on the operator. The 15kb estimate is for an operator used to calculate an incrementally updated average window. Generally, the memory consumption is linear to the number of operators in the graph. For special operators, e.g., storage operators as the multidimensional histogram, the memory size will increase during runtime. For the multidimensional histogram however, even for a large number of dimensions (11), the consumed memory of the trie structure and its content stayed well under a megabyte within one day's use.

[9]Experiments have shown that (for a typical Mercedes-Benz midsize luxory sedan) the average bus load was 33.7%. The engine CAN bus operates at 500kbit/s.

VI. Conclusions

The current automobile's electronic architecture is influenced by many different standards. There are ambitions within the automotive industry to standardize the diagnostic software interface in a way that abstract data sources such as "vehicle speed" are mapped to concrete sensors on the fly [27]. This would ease data acquisition. The big advantage of using the vehicle's diagnostic infrastructure is that there is virtually no need to modify the car. Therefore, the system is easy to install and remove. Our system employs these benefits and combines them with advanced recording strategies.

Our work shows that given interfaces—OBD-II and diagnostic protocols—can be used to achieve a high-level goal by employing low-level tools and data. We show how a flexible and adaptable data recorder can be used in various vehicle-related contexts and for different purposes, by using one standardized interface and a small amount of processing power and memory. Our situation-dependent recording demonstrates a great improvement over existing systems and allows a selective reduction of data quantity while maintaining the adequate data quality. Engineers can start out with a generic configuration and subsequently add filter and aggregate operators, refine recording operations and actions on specific events. To leverage root-cause analysis for detecting and diagnosing unforeseen situations, our system could be enhanced by operators implementing clustering algorithms. The goal of our system design, however, has been to support engineering testing, so that engineers can use a-priori knowledge. Therefore, it is difficult to directly compare recorded data volume to other systems, as it varies with the system's configuration and use. The prototype system has been tested at research and development facilities in North America and Germany.

In the future, stream processing in the vehicle can be enhanced with more advanced operators, which can employ data mining and machine learning techniques on multiple data-streams and will be able to show correlations in case of errors. For example, multi dimensional state space clustering and learning allows to define different states of the vehicle (e.g., clean combustion and normal acceleration behavior vs. unusual behavior in acceleration or braking) and detect unusual patterns automatically. By using such performance-intensive algorithms, one can use the processing power of the on-board unit to full capacity. This may lead to additional requirements, such as scheduling operator execution in a quality-of-service based manner, as it was already examined for different (non-embedded) stream processing applications. As bandwidth volume for transmission is limited, prioritization of specific, e.g., recent data, with regard to granularity of saved probes may be applied.

We believe that this area of research, intelligent preprocessing and condensation of possibly remote sensor data, will significantly gain attention as electronic automotive systems increasingly assist the classical mechanical domain. The electronic systems of a car provide valuable information

about the vehicle's state to engineers, workshops and quality assurance—data only need to be processed, preserved, and evaluated appropriately.

REFERENCES

[1] J. Fröberg, K. Sandström, C. Norström, H. Hansson, J. Axelsson, and B. Villing, "Correlating bussines needs and network architectures in automotive applications - a comparative case study," in *Proceedings of the 5th IFAC International Conference on Fieldbus Systems and their Applications (FET)*. Aveiro, Portugal: IFAC, July 2003, pp. 219–228.

[2] H. Schweppe, A. Zimmermann, and D. Grill, "Flexible in-vehicle stream processing with distributed automotive control units for engineering and diagnosis," *Industrial Embedded Systems, 2008. SIES 2008. International Symposium on*, pp. 74–81, June 2008.

[3] S. Warren, "In-vehicle data logging," *Embedded Linux Journal*, vol. 5, pp. 14–16, 18–19, Sep,Oct 2001.

[4] S. Ilic, J. Katupitiya, and M. Tordon, "In-vehicle data logging system for fatigue analysis of drive shaft," in *International Workshop on Robot Sensing, 2004. ROSE*, May 2004, pp. 30–34.

[5] V. Barabba, C. Huber, F. Cooke, N. Pudar, J. Smith, and M. Paich, "A multimethod approach for creating new business models: The General Motors OnStar project," *Interfaces*, vol. 32, no. 1, pp. 20–34, 2002.

[6] K. Grimm, "Software technology in an automotive company: major challenges," in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 498–503.

[7] ISO/IEC IS, "ISO 14230:2000(E) – Road vehicles – Diagnostic systems – Keyword Protocol 2000," *TC 22/SC 3, International Organization for Standardization, Geneva, Switzerland*, 2000.

[8] ISO/IEC IS, "ISO 15765-2:2004(E) – Road vehicles — Diagnostics on Controller Area Networks (CAN) — Part 2: Network layer services," *TC 22/SC 3, International Organization for Standardization, Geneva, Switzerland*, 2004.

[9] L. Golab and M. T. Özsu, "Issues in data stream management." *SIGMOD Record*, vol. 32, no. 2, pp. 5–14, 2003.

[10] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. F. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. B. Zdonik, "Retrospective on Aurora." *VLDB J.*, vol. 13, no. 4, pp. 370–383, 2004.

[11] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik, "The design of the Borealis stream processing engine." in *CIDR*, 2005, pp. 277–289.

[12] N. Tatbul and S. B. Zdonik, "Window-aware load shedding for aggregation queries over data streams." in *VLDB*, U. Dayal, K.-Y. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y.-K. Kim, Eds. ACM, 2006, pp. 799–810.

[13] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom, "STREAM: The Stanford stream data manager." *IEEE Data Eng. Bull.*, vol. 26, no. 1, pp. 19–26, 2003.

[14] S. Madden, R. Szewczyk, M. J. Franklin, and D. E. Culler, "Supporting aggregate queries over ad-hoc wireless sensor networks," in *WMCSA*. IEEE Computer Society, 2002, pp. 49–58.

[15] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah, "TelegraphCQ: Continuous dataflow processing for an uncertain world." in *CIDR*, 2003.

[16] J. Considine, F. Li, G. Kollios, and J. W. Byers, "Approximate aggregation techniques for sensor databases," in *ICDE*. IEEE Computer Society, 2004, pp. 449–460.

[17] H. Kargupta, R. Bhargava, K. Liu, M. Powers, P. Blair, S. Bushra, J. Dull, K. Sarkar, M. Klein, M. Vasa, and D. Handy, "Vedas: A mobile and distributed data stream mining system for real-time vehicle monitoring." in *SDM*, M. W. Berry, U. Dayal, C. Kamath, and D. B. Skillicorn, Eds. SIAM, 2004.

[18] Encirq Corporation, "Automotive stream-based data management," December 2006, encirq Corporation, 577 Airport Boulevard, Suite 700, Burlingame, CA 94010-2024, available at http://www.microcontroller.com/Embedded.asp?did=154 [last accessed 21-Jun-2009].

[19] H. Gomaa, "A software design method for real-time systems." *Commun. ACM*, vol. 27, no. 9, pp. 938–949, 1984.

[20] M. Broy and G. Ştefănescu, "The algebra of stream processing functions," *Theoretical Computer Science*, vol. 258, no. 1–2, pp. 99–129, 2001.

[21] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik, "Monitoring streams - a new class of data management applications." in *VLDB*. Morgan Kaufmann, 2002, pp. 215–226.

[22] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik, "Aurora: A new model and architecture for data stream management." *VLDB J.*, vol. 12, no. 2, pp. 120–139, 2003.

[23] S. Krishnamurthy, C. Wu, and M. J. Franklin, "On-the-fly sharing for streamed aggregation." in *SIGMOD Conference*, 2006, pp. 623–634.

[24] R. Love, "Introducing the 2.6 kernel," *Linux Journal*, vol. 2003, no. 109, p. 2, 2003.

[25] R. Sedgewick, *Algorithms, 2nd Edition*. Addison-Wesley, 1988.

[26] U. Kiencke and L. Nielsen, *Automotive Control Systems, for Engine, Driveline, and Vehicle*, 2nd ed. Springer Verlag, 2005.

[27] ASAM, "Association for standardisation of automation and measuring systems," founded December 1998, http://www.asam.net/ [last accessed 21-Jun-2009].

Hendrik Schweppe received his Dipl.-Ing. degree from Technische Universität Berlin in 2008. He is active in vehicle research since 2005, when he joined DaimlerChrysler Research Berlin. His diploma thesis was performed during a research stay at Mercedes-Benz RDNA in Palo Alto, USA, where he designed and prototyped a new in-vehicle stream processing system. At Daimler's research center in Ulm, Hendrik visited the data mining and quality engineering research group during summer 2007. In March 2008 he joined PacketVideo Germany GmbH, a young Fraunhofer spin-off, where he worked on mobile device integration of DLNA software and vehicle specific aspects of multimedia networks and participated in the DLNA automotive task-force. He joined the research institute EURECOM in Sophia-Antipolis, France, in June 2009, where he is going for his Ph.D. He is member of the networking and security group, where he works on in-vehicle security systems, with a focus on on-board communication. He is involved in the EVITA EU project. His research interests include distributed systems, automotive and embedded systems as well as security. Mr. Schweppe is a member of Gesellschaft für Informatik.

Armin Zimmermann (M'08) computer science Dipl.-Inform. degree '93, Dr.-Ing. degree '97, Habilitation '07 Since 1994 he was Research Assistant and later PostDoc with the performance evaluation group at TU Berlin, and has been project manager of the TimeNET software tool since '96. During this time he was the principal investigator of the research group Model-Based Evaluation of Discrete Real-Time Systems and coordinated the graduate research training group "Stochastic Modelling and Quantitative Analysis of Complex Systems in Engineering". His Ph.D. thesis on modelling and analysis of manufacturing systems with Petri nets received three awards. In March 1998 and from September to December 1999 he was a visiting scientist at the Universidad de Zaragoza, Spain and in 2001 he was guest lecturer at Shanghai Jiao Tong University. He was a guest professor at Hasso Plattner Institute for IT Systems Engineering and held a substitue professorship for Real-Time Systems and Robotics at TU Berlin 2006 - 2008. Since then, he heads the System and Software Engineering group at Technische University Ilmenau, Germany, in the Computer Science and Automation faculty. He published a book on Stochastic Discrete Event Systems (Springer 2007). His research interests include modelling, performance evaluation, optimization, and control of technical systems using discrete-event models as well as their tool support.

Prof. Zimmermann is a member of GI and DHV. He serves as Associate Editor of IEEE Transactions on Industrial Informatics and is a member of the Industrial Automated Systems and Controls Subcommittee of the IEEE IES Technical Committee on Factory Automation.

Daniel Grill is Group Manager at Mercedes-Benz Research and Development North America, Inc, located in Palo Alto, CA, USA. He is responsible for advanced development and development of in-vehicle Internet based services and vehicle relationship management applications for Mercedes-Benz passenger cars. During the past 10 years he introduced multiple features and process innovations to Mercedes-Benz and Chrysler products and development processes. Prior, he worked as research scientist in the topic area of Mobile Computing at the Daimler-Benz Research Center in Ulm, Germany. He holds a diploma in Electrical Engineering from University of Stuttgart, Germany.