TruckSTM: Runtime Realization of Operational State Transitions for Medium and Heavy Duty Vehicles

SUBHOJEET MUKHERJEE, JEFFREY C. VAN ETTEN, NAMBURI RANI SAMYUKTA, JACOB WALKER, INDRAKSHI RAY, and INDRAJIT RAY, Colorado State University

Embedded computing devices play an integral role in the mechanical operations of modern-day vehicles. These devices exchange information containing critical vehicle parameters that reflect the current state of operations. Such information can be captured for various purposes, such as diagnostics, fleet management, and analytics. Although monitoring individual parameters can be useful for some applications, monitoring distinct combinations of parameters can reveal more complex and higher-level states that may give useful information. Existing monitoring systems either lack user configurability and control or present simple user interfaces that make it difficult to monitor and collate different parameters to observe high-level vehicle states. In this work, we present TruckSTM, a novel application that realizes user-defined states from messages seen in the embedded networks of medium and heavy duty vehicles and displays state transitions on an interactive user interface. We begin by symbolically formulating some of the in-vehicle networking concepts and formally defining the concept of operational states and state transitions. We then elaborate on the operations performed by TruckSTM in mapping network-obtained vehicle parameters to states that can be defined in standard JSON format. Finally, we evaluate TruckSTM's asymptotic performance and present the results for the worst-case scenario and demonstrate that in a real world scenario such high level state visualization constraints of an operational truck.

CCS Concepts: • **General and reference** → **Design**; **Performance**; • **Networks** → **Cyber-physical networks**; • **Human-centered computing** → **Visualization**; • **Software and its engineering** → **Designing software**; • **Theory of computation** → *Design and analysis of algorithms*;

Additional Key Words and Phrases: Automotive networks, SAE J1939, operational state, transition, visualization

ACM Reference format:
Subhojeet Mukherjee, Jeffrey C. Van Etten, Namburi Rani Samyukta, Jacob Walker, Indrakshi Ray, and Indrajit Ray. 2019. TruckSTM: Runtime Realization of Operational State Transitions for Medium and Heavy Duty Vehicles. *ACM Trans. Cyber-Phys. Syst.* 4, 1, Article 4 (October 2019), 25 pages.
https://doi.org/10.1145/3300183

4

1 INTRODUCTION

Most existing transportation management systems lack sophisticated real-time monitoring capabilities of "custom-defined high-level" states of a vehicle or a fleet that are the result of correlating

The work was supported in part by the NSF under award numbers CNS 1619641 and CNS 1715458.

Authors' addresses: S. Mukherjee, J. C. Van Etten, N. R. Samyukta, J. Walker, I. Ray, and I. Ray, Colorado State University, 1873 Campus Delivery, Fort Collins, CO, 80523; emails: {subhomuk, jvanette, samyu}@colostate.edu, jksctwkr@rams.colostate.edu, {Indrakshi.Ray, Indrajit.Ray}@colostate.edu.

2378-962X/2019/10-ART4 $15.00
https://doi.org/10.1145/3300183

and fusing different observable lower-level states. Examples of these custom-defined high-level states include those related to mundane driving conditions, such as acceleration and braking, and more complex ones such as those related to vehicular security and safety. Increasingly, various stakeholders are perceiving the need for such capabilities. For example, with the growing interest in heavy vehicle security [18], a fleet company is interested in monitoring potentially malicious and intrusive actions like un-warranted disabling of engine brakes at low speeds [1]. State-based systems and metrics have been identified as important in the vehicle security [4] and safety [19] research communities. The ability to observe and act upon custom-defined high-level states enables faster and better decision making. Observing states may allow users to create an intrusion detection system that generates actionable alerts when specific security critical states occur. In this work, we present TruckSTM, a novel application that realizes user-defined states from messages seen in the embedded networks of medium and heavy duty vehicles and displays state transitions on an interactive user interface.

Diagnostic and telematics data are collected from in-vehicle communication (bus) networks that typically follow the CAN [6] protocol. Embedded computers, referred to as Electronic Control Units (ECUs), transmit and receive messages on the CAN buses. These computers control a variety of operations ranging from critical ones, such as fine-tuning fuel injection, to non-critical ones, such as comfort- and entertainment-related functions. To make better and more informed decisions, ECU applications require timely and accurate information from other ECUs. This information is received over the CAN network and can also be accessed by on-board devices connected directly or indirectly to these networks. Medium and heavy duty vehicle information on the CAN bus adhere to the specifications made in the SAE J1939 [11] standards. This allows for interoperability of components supplied by different Original Equipment Manufacturers (OEMs). TruckSTM is developed keeping the SAE J1939 standards in mind. In this work, we formalize the concepts of heavy-vehicle states and identify the practical challenges associated with designing and implementing a real-time system that realizes high-level user-defined states. To achieve this goal, we have designed a Linux-based application that receives parameters (engine speed, torque, switch states, etc.) from in-vehicle network traffic, maps them to user-defined states, and displays state transitions—all in real-time. For example, our system can map the engine requested torque and the current vehicle speed to the security critical state "`disabling engine braking by commanding 0% engine torque at low vehicle speed`" [1] and show the transitions to and from this state.

J1939 recommends the usage of the high-speed CAN protocol for physical communication. Although high-speed CAN permits up to 1 Mbits/sec of transmission speed, J1939 standards recommend a speed of 250 kbits/sec [14, 16]. In a real-time system, every message should be considered for evaluation. On a 250-kbits/sec bus, a typical J1939 application message can be seen every 0.5 ms [11]. We refer to this time bound as the *soft timing constraint* as, in general, owing to several factors like ECU processing capabilities, bus contention, J1939 standards, and mechatronic behavior, the real message inter-arrival times are much higher. As a matter of fact, the soft constraint on processing time is seldom realized. In one of our previous works [3], we used two real-world heavy vehicle network logs for experimentation purposes. For both of these logs, we observed average message inter-arrival times of around 3 and 5 ms. For the purpose of this article, we consider these time limits to be the *hard timing constraint*s. TruckSTM's real-time capabilities are designed to operate appropriately within these timing constraints.

We make several contributions in this article. We present a detailed overview of the J1939 application layer standards using symbolic formulation. We define the concept of an *operational state*[1] for heavy vehicles and then realize the definition using a JSON schema. We present TruckSTM,

[1]The terms *state* and *operational state* refer to the same entity.

which realizes vehicle states and transitions from in-vehicle network traffic. We demonstrate the collaborative workflow of TruckSTM components and evaluate worst-case performance characteristics using asymptotic analysis of individual components. We also perform a detailed evaluation of TruckSTM's runtime performance in the worst case.

The rest of the article is organized as follows. In Section 2, we describe related effort in this area. In Section 3, we provide the relevant background information. In Section 4, we describe the implementation and performance-related aspects of TruckSTM. In Section 5, we perform a detailed performance evaluation that tests the ability of TruckSTM to meet the timing constraints in the worst case. In Section 6, we conclude the article with pointers to future work.

2 RELATED WORK

Some works [5, 8] have been done on the design of systems and infrastructure for acquisition and (remote) dissemination of information from in-vehicle networks. Some works also describe techniques for visualization and intelligent utilization of acquired vehicle parameters [7, 20]. However, existing literature does not address the problem of formalizing and realizing user-defined states from in-vehicle network traffic. The definition for user-defined states presented in this article allows users to visualize states as a combination of various parameter values, but also as single-parameter instances. For example, one can visualize different states of the braking switch and engine speed individually or in combination. Furthermore, states generated in this manner can be readily used for further processing and visualization.

Vector's[2] CANOe tool [17] provides the *State Tracker* (ST) application that is closest to TruckSTM. Vector provides a tutorial video[3] that describes the operations supported on the ST window. The ST application allows users to track active states and transitions for individual vehicle parameters as pulses or rectangular blocks moving across an interactive user interface. Users can configure the following in the user interface. (i) Set triggers on certain parameter values and pause/freeze the window. Multiple triggers can be active in parallel, but only a single trigger pauses the window. This means that states are not activated by parameter combinations but only by single-parameter instances. (ii) Set colors to specific parameter ranges. For example, engine speed greater than 2,000 can be configured to be displayed using the color red. Since the ST window displays multiple parameters in one window, it is possible to display different colors and raise alarms when particular color combinations occur. For example, if high engine speed and low gear are both displayed in red, this can symbolize potential clutch damage. However, similar color combinations will also have to be set for other states. The issue with this approach is that if every parameter range in a state is displayed using the same color, the same parameter range cannot be included in two states, as in that case multiple colors will correspond to the same state. In combination with the sheer speed and volume of information flowing across the ST window, this can make it extremely difficult for the human eye to perceive minute details in a timely fashion, thus resulting in possibly erroneous judgments.

Our system does not require any human intervention and deals with rapidly arriving network traffic efficiently to realize state changes. Although some visual changes can be rapid, even acquiring encoded but critical information like active states and transitions can be of great use to many applications that require such information for further processing. Moreover, commercial tools are expensive, and integration may require effort. It is also not possible to customize and extend the tool due to the absence of source code. Our effort demonstrates how such a tool can be built and the challenges associated with developing one.

[2]https://www.youtube.com/watch?v=QZfWD4wewV4.
[3]https://www.youtube.com/watch?v=ozp2up-UhcA&t=325s.

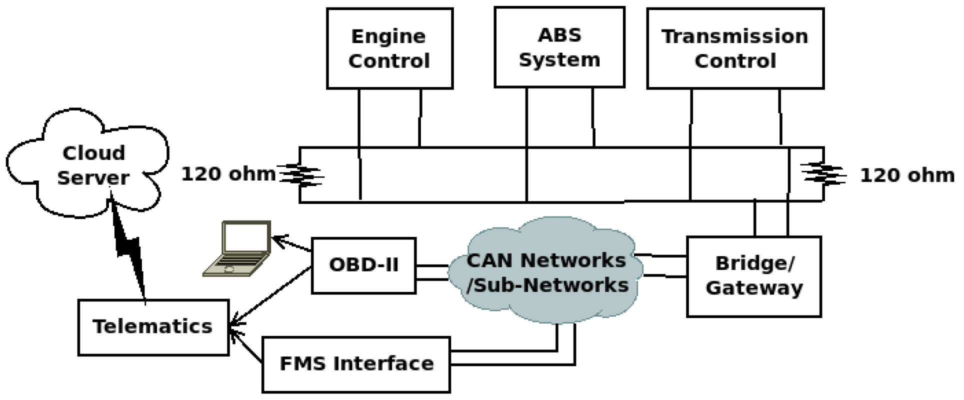


Fig. 1. In-vehicle networking and external interfaces.

3 FORMALIZATION OF IN-VEHICLE COMMUNICATION

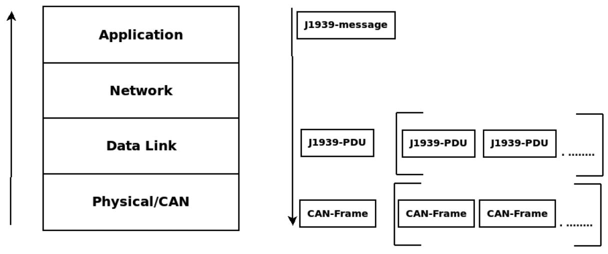
3.1 In-Vehicle Communication Overview

Figure 1 shows a typical J1939-based network organization [10, 11]. Information transmitted by an ECU on a physical bus can be received by all other ECUs on the same *segment.* A single segment or multiple segments connected via *bridges* or *gateways* form a J1939 network. For heavy duty trucks, trailer networks/sub-networks are often connected to the main tractor network using bridges or gateways that can filter messages across different networks depending on bandwidth and application requirements. For example, if the instrument cluster needs to show the status of trailer lights, it may request (or receive) such information from the trailer network via a bridge or gateway.

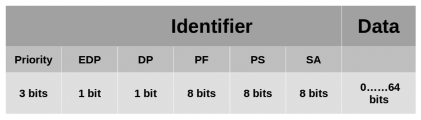
The powertrain network from Figure 1 includes an engine controller, a transmission controller, and an anti-lock braking system (ABS). For example, in some systems, under a tire skid scenario, the ABS may request retardation to be disabled by sending messages to the engine control module. Since multiple ECUs communicate using the same physical media, timely delivery of such messages demands that messages are not delayed or lost due to collision. For this purpose, SAE J1939 standards prescribe the usage of the high-speed CAN protocol for physical transmission.

CAN [6] is a multi-master serial bus protocol that facilitates highly reliable message transmission over a two-wire physical medium (as shown in Figure 1). The CAN protocol operates using a CSMA/CD-like scheme with priority-based arbitration. In such a scheme, ECUs transmit messages when a bus is idle, and if two ECUs attempt to transmit at the same time (i.e., collision is detected), the message with higher priority wins the arbitration process. The ECU transmitting the lower-priority message waits until the next idle slot is available. For example, the ABS system transmits messages with high priority, thereby ensuring necessary control of the bus during safety-critical scenarios. Moreover, high-speed CAN allows significant transmission speeds of up to 1Mbits/sec. ECUs can thus receive and react to critical alerts in a timely fashion. Additionally, CAN also facilitates efficient error management and recovery, thus making it the primary choice for modern in-vehicular communication.

CAN ensures reliable message delivery across ECUs but does not specify how messages must be interpreted. SAE J1939 fills this gap by specifying standards that may be opted by different manufacturers for reliable communication between heterogeneously manufactured ECUs. This common set of standards also allows effective data logging and analysis (Figure 1). Such analysis can be performed by logging J1939 network traffic and processing traffic data in either offline fashion using



(a) J1939 Protocol Layers



(b) J1939 Protocol Data Unit

Fig. 2. SAE J1939 networking concepts.

onboard connectors like the OBD-II port or online via standardized Fleet Management System (FMS[4]) interfaces and/or telematics units.

3.2 SAE J1939

3.2.1 A Layered Networking Paradigm. The SAE J1939 standards [11] are modeled on the seven-layer ISO/OSI networking protocol stack. Currently, only four of the seven layers are documented as shown in Figure 2(a). Unlike the ISO/OSI paradigm, messages are not strictly encapsulated with new headers at each layer. Instead, functionalities at different layers are invoked when required. For example, every node on the J1939 network has a unique address, and when nodes on the network claim addresses, *network* layer functionalities are invoked.

Raw signal values switch states and other vehicle parameters are encoded and packaged in J1939 messages following specifications made in the application layer documentation (SAE J1939-71 [15]) and the J1939 Digital Annex (DA) [9]. The network-layer (SAE J1939-31 [12]) describes procedures and devices (like bridges and gateways) that enable seamless networking within and across segments. The data-link-layer specifications (SAE J1939-21 [13]) enumerate the various message types and describe protocols and algorithms that ensure reliable message delivery. The physical/CAN layer is responsible for physical communication. It is at this layer that messages are structured as CAN frames and transmitted on the bus.

3.2.2 Message Interpretation. J1939 messages are composed at the application layer. A single CAN frame allows a maximum of 64 bits of data. J1939 allows larger messages by splitting (Figure 2(a)) at the data-link layer into J1939 Protocol Data Units (PDUs) that are then transmitted sequentially and reliably to the receiving ECU(s). PDUs can be split into 29-bit identifiers and 0-to-64-bit data fields as shown in Figure 2(b). The identifier field can be further split into six distinct fields, namely priority, extended data page (EDP), data page (DP), PDU format (PF), PDU specific (PS), and source address (SA). The first 3 bits of a J1939 PDU denote the relative criticality of a message and aid in the process of arbitration. EDP and DP are 1-bit values and can together assume only a pair of standardized values 00_2 and 01_2 (we use the suffix to denote the base of the number). We use lowercase letters to denote the values of these fields. For example, ps_{16} denotes the value of the PS address in hexadecimal form. Table 1 shows a summary of the symbols used in the article.

J1939 identifies each vehicle parameter using a unique identifier, the Suspect Parameter Number (SPN). Parameters can be grouped together according to one or more J1939 standardized convention [11]. These groups are uniquely identified by an 18-bit number referred to as Parameter Group Numbers (PGNs). Every message has a unique PGN. When a message is received, the PGN is

[4]http://www.fms-standard.com/.

Table 1. Summary of Important Symbols

Symbol	Explanation	Reference
dp	Data page field of message	3.2.2
edp	Extended data page field of a message	3.2.2
pf	PDU format field of a message	3.2.2
ps	PDU-specific field of a message	3.2.2
pgn	Parameter Group Number, a unique identifier for a message	3.2.2
da	Destination address of a message	3.2.2
sa	Source address of a message	3.2.2
spn	Suspect Parameter Number, a unique id for a parameter encoded in a message	3.2.2
Pn	Numeric parameter	3.2.2
V_{Pn}	Value of a numeric parameter	3.2.2
$\mathcal{V}_{Pn}$	Universal J1939 numeric parameter range	3.2.2
Pa	ASCII parameter	3.2.2
V_{Pa}	Value of an ASCII parameter	3.2.2
$\mathcal{V}_{Pa}$	Universal J1939 ASCII parameter range	3.2.2
Pb	Binary parameter	3.2.2
V_{Pb}	Value of a binary parameter	3.2.2
$\mathcal{V}_{Pb}$	Universal J1939 binary parameter range	3.2.2
p	A parameter (Pn or Pa or Pb), identified by the tuple (pgn, da, sa, spn)	3.3.1
$pinst$	An instance of a parameter in a state definition	3.3.2
$pinstSet$	A set of parameter instances	3.3.3
s	A vehicle state $\in S$ expressed as $\{\langle p, pinstSet \rangle\}$	3.3.3
M_t	Mode $\in \mathcal{M}$ of a vehicle at time t	3.3.3
$\mathcal{T}$	Set of all mode transitions, where a mode transition is $(M_{t_i}, M_{t_{i+1}})$	3.3.3

obtained from the PF and PS fields as follows:

$$pgn_{16} = \begin{cases} (edp_2 dp_2)_{16} pf_{16} 00_{16} & \text{if } pf < 240 \\ (edp_2 dp_2)_{16} pf_{16} ps_{16} & \text{otherwise} \end{cases}$$

It must be noted that when $pf < 240$ and $dp = 0$, then $pgn < 61440$ and $pgn_{16} < F000$ as $edp = 0$. If $pf < 240$ and $dp = 1$, then $pgn < 126976$ or $pgn_{16} < 1F000$ as $edp = 0$.

CAN is a broadcast medium, but J1939 also allows destination-specific communication. Thus, every controller application connected to the bus must have a unique 8-bit address, some of which are standardized by J1939. For example, the engine controllers are assigned standard addresses of 00_{16} and 01_{16}. Source and destination addresses are also embedded in the identifier field of the PDU. The address of the sending device is in the SA field. The destination address is obtained as follows:

$$da_{16} = \begin{cases} ps_{16} & \text{if } pf < 240 \\ FF_{16} & \text{otherwise} \end{cases}$$

In other words, an ECU can send a PGN with PF less than 240 by embedding the destination address in the PS field. Otherwise, the message is a broadcast and all nodes on the network can process it.

Once a PGN is identified, SPNs grouped under that PGN can be used to interpret the contents of the *data* field. Each SPN is assigned to a set of attributes, namely length, position, resolution, and offset, for this purpose. Data bits are first extracted according to the given *position* and *length*.

Extracted bits are then converted into parameter values using the *resolution* and *offset* factors and expressed in *unit*s. The following example demonstrates one such scenario:

$$id \rightarrow \underbrace{110}_{p}\;\overbrace{\underbrace{0}_{edp}\;\underbrace{0}_{dp}\;\underbrace{11111101}_{pf}\;\underbrace{11100001}_{ps}}^{pgn}\;\underbrace{00110001}_{sa}$$

$$data \rightarrow \underbrace{01100100}_{spn2609}\,111111\;\underbrace{01}_{spn7853}\;11111111111111..$$

Since the value contained in the PF field is greater than 240, the PGN is calculated using all EDP, DP, PF, and PS fields. In this case, the binary to hexadecimal conversion yields a PGN FDE1 (64993_{10}), which denotes "Cab A/C Climate System Information" [9]. This PGN is associated with two distinct SPNs [9]:

- ***2609***: "Cab A/C Refrigerant Compressor Outlet Pressure," expressed in byte 1 (from left) and calculated by multiplying the decimal equivalent with the scaling factor 16 (resolution of "16kPa/bit") and adding to it an offset of 0.
- ***7853***: "Air Conditioner Compressor Status," expressed in byte 2 (from left) bits 0 to 1 and used in binary form.

The actual value of the "Cab A/C Refrigerant Compressor Outlet Pressure" can thus be calculated as $01100100_2 * 16_{10} + 0_{10} \rightarrow 100_{10} * 16_{10} \rightarrow 1600\ kPa$. Similarly, the "Air Conditioner Compressor Status" is set to 1, which denotes that the "Air Conditioner Compressor is ON" [9].

Application layer standardized parameter (spn) types and associated resolutions, scaling factors (r), and offsets (o) are as follows:

Numeric: Resolutions are expressed as "r *unit* /bit" ("r *unit* per bit"), where $r \in \mathbb{R}_{>0}$ (r is a positive real number) and *unit* can be measurements like "rpm" and "kPa." An example resolution can be "16 kPa/bit." Offset $o \in \mathbb{R}_{\leq 0}$ is used to express zero or negative parameter values. We denote a numeric parameter by the notation Pn and its value by V_{Pn}, where $V_{Pn} = parameterBits_{10} * r + o$ (also shown in the preceding example). Thus, $o \leq V_{Pn} \leq (2^l - 1) * r + o$, where l is the number of bits in the parameter.

Finally, we define the universal numerical J1939 parameter range as $\mathcal{V}_{Pn} = \bigcup_{Pn} V_{Pn}$.

ASCII: Resolutions are expressed in "ASCII." No scaling factor and offset are required. Parameter values are ASCII strings like VIN numbers. We use the notation Pa to an ASCII parameter of bit length l. V_{Pa}, the value of Pa, is thus an $l/8$ characters long ASCII string $c_1c_2..c_{l/8}$, where $c \in$ ISO Latin-1 character set. Finally, we define the universal ASCII J1939 parameter range as $\mathcal{V}_{Pa} = \bigcup_{Pa} V_{Pa}$.

Binary: Resolutions are expressed as "Binary" or "l bit bit-mapped," where l is the parameter bit length. No scaling factor and offset are required. Parameter values are bit strings of length l and can be expressed as l-tupe bits. For example, "010" can be expressed as $(0, 1, 0)$. We denote a binary parameter as Pb and its value as V_{Pb}, where $V_{Pb} \in \{0, 1\}^l$. Finally, we define the universal numerical J1939 parameter range as $\mathcal{V}_{Pb} = \bigcup_{Pb} V_{Pb}$.

3.3 Operational States, Modes, and Mode Transitions

3.3.1 Identifying Parameters. A parameter can be uniquely identified using the SPN number. However, J1939 allows the same parameter to be transmitted by multiple sources [15]. The standards do not prevent a parameter from being assigned to multiple groups [10]. Moreover, parameters with the same PGN and transmitted by the same source can be targeted for different destinations. This is especially true for control and command messages. Accordingly, a parameter on a bus can be uniquely identified by the collection (pgn, da, sa, spn), and we denote it using the symbol p. An example parameter on the CAN bus can thus be represented as $p = (61444, 255, 0, 190)$.

3.3.2 Parameter Values Discretization. A single state can be associated with different parameter values. For example, in an idle situation, a very low engine speed or a very high speed can both denote a state of malfunction. Thus, to define a state of malfunction, one has to include both low and high RPM values. However, as already seen in Section 3.2.2, parameters can attain different number of values ($\in \mathcal{V}_{Pn} \cup \mathcal{V}_{Pa} \cup \mathcal{V}_{Pb}$) depending on their lengths. Parameters can have small bit lengths if they are of an enumerated type, such as those expressing switch states. Other parameters having continuous values, such as engine speed, require larger bit lengths. Thus, the term *low* engine speed can signify a large number of values, which need to be discretized into ranges to avoid state explosion [19]. We achieve this by allowing parameter values *pinst* to be expressed as closed intervals:

$$pinst = \begin{cases} [k - l] & \text{if } k, l \in \mathcal{V}_{Pn} \land (k, l \notin \mathcal{V}_{Pa} \cup \mathcal{V}_{Pb}) \\ k & \text{otherwise} \end{cases}$$

Binary and ASCII parameters are discrete by their very nature. Numeric parameters are explicitly discretized as earlier.

Note that every parameter value is associated with a parameter. In other words, $pinst = [0.0 - 1100.0]$ is a range of values for some parameter, say $p = (61444, 255, 0, 190)$.

3.3.3 Combining Parameters into Operational States. Zhang et al. [19] define a state as a category or collection of discrete/discretized instances of a single parameter. Such a definition suits the anomaly detection requirements from Zhang et al. [19] but fails to convey a more high-level view of the vehicle's behavior. For example, being at a state of "high RPM" alone does not convey a state of "possible clutch damage" unless it co-exists with another state of "low gear." Thus, combining parameter instances from different parameters into a single state can help capture a broader aspect of operations. A state definition that includes both "high RPM" and "low gear" can be clearly termed as a state of "possible clutch damage." With this view, we define operational states as follows.

Definition 1 (Operational State). An operational state for medium and heavy duty vehicles denotes the behavior of the vehicle during a period of driving as a unique combination of the vehicle's operational parameter values. The operational state s consists of a set of pairs of the form $< p, pinstSet >$ as defined in the following.

$s = \{\langle p, pinstSet \rangle\}$, where $pinstSet$ is the set of values of parameter p when the state is s.

We use $\mathcal{S}$ to denote the set of all operational states. Every defined state is thus a combination of a set of parameter values where every parameter value can be expressed as a closed interval. Note that we do not require individual intervals to be disjoint. Using the definition from earlier, the state of "possible clutch damage" can possibly be represented using parameters (190 and 523) transmitted by "engine" and "transmission," respectively: $\{\langle (61444, 255, 0, 190), \{[3000 - 6000], [6000 - 8000]\}\rangle, \langle (61445, 255, 3, 523), \{[0, [1 - 50]\}\rangle\}$.

Definition 2 (Active Parameter). A parameter p_i with value v_i is said to be active in some operational state s if $\exists \langle p, pinstSet \rangle \in s$ such that $p_i = p$ and $v_i \in pinstSet$.

Definition 3 (Affected State). An operational state s is said to be affected if it has one or more active parameters in incoming messages during a given duration.

Definition 4 (Active State). An operational state s is said to be active if all of its parameters are active in incoming messages during a given duration.

A system can be in several active operational states at any given time. We formalize this using the concept of mode.

Definition 5 (Mode). A mode M_t is a set of operational states that can be simultaneously active at time t.

$$M_t = \{s\} \text{ where } s \in \mathcal{S}$$

We use $\mathcal{M}$ to denote all of the modes. Mode transitions occur with the passage of time. When mode transitions occur, one or more operational states change and others remain the same.

Definition 6 (Mode Transition). A mode transition t for medium and heavy duty vehicles defines a shift from an active set of states at a given time instance to a new set of states at the next time instance. Mode transitions are defined as a pair of modes $(M_{t_i}, M_{t_{i+1}})$ defined in the following.

$$(M_{t_i}, M_{t_{i+1}}), \text{ where } i \in \mathbb{Z} \text{ and } M_{t_i}, M_{t_{i+1}} \subseteq \mathcal{M} \text{ and } |M_{t_i} \cap M_{t_{i+1}}| \geq 0$$

According to the preceding definition, a vehicle can be in more than one active operational state at any given time and transitions can involve overlapping states. Let us consider the simple braking and anti-lock braking states. These are overlapping states—that is, when heavy braking occurs, anti-lock braking also occurs. Furthermore, at the next time instant, ABS can co-exist with some other state even though braking has been stopped. One important observation from this is that modes and transitions can together form a mode transition graph $\mathcal{G} = \langle \mathcal{M}, \mathcal{T} \rangle$, where $\mathcal{M}$ represents the set of all modes and $\mathcal{T}$ the set of all mode transitions. The mode transition graph $\mathcal{G}$ is a directed graph. An edge (M_i, M_j) signifies that there is a transition from mode M_i to mode M_j. Drawing graphs on the screen, especially determining correct layouts, is a difficult problem. Thus, in TruckSTM, we represent edges using color combinations. We use the color `cyan` to denote *transitional* or previously active states $(M_i \setminus M_j)$ and the colors `green` or `red` (refer to Section 4.2.5) to denote currently active states (M_j). The inactive states at a given time $(S \setminus M_j \cup M_i)$ are represented using the color `grey`.

4 SOLUTION METHODOLOGY

4.1 Overview of Solution and Challenges Addressed

We make use of two example vehicle operational states that are of interest to the user:

***Sudden high acceleration demand*:** This is an abnormal state that reflects a sudden high press on the accelerator pedal that activates the kickdown switch. The state is represented using the following three parameter instances:

- $\{\langle(61443, 255, 0, 91), \{[70.0 - 90.0], 100.0\}\rangle\}$ High pedal press percentages (70.0% to 90.0% and 100.0%) broadcasted by the engine controller in the SPN 91 from the PGN 61443.
- $\{\langle(61443, 255, 0, 559), \{1.0\}\rangle\}$ Kickdown switch active (1.0) confirmation broadcasted by the engine controller in the SPN 559 from the PGN 61443.
- $\{\langle(65265, 255, 0, 84), \{[0.0 - 40.0]\}\rangle\}$ Low wheel-based vehicle speed (0.0 to 40.0 mph) broadcasted by the engine controller in the SPN 84 from the PGN 61443.

***Engine brake disable attack*:** This state is adopted from an attack described in Burakova et al. [1]. The state represents malicious disabling of engine brakes by commanding a 0% torque when the vehicle is actively decelerating. The state is represented using the following two parameter instances:

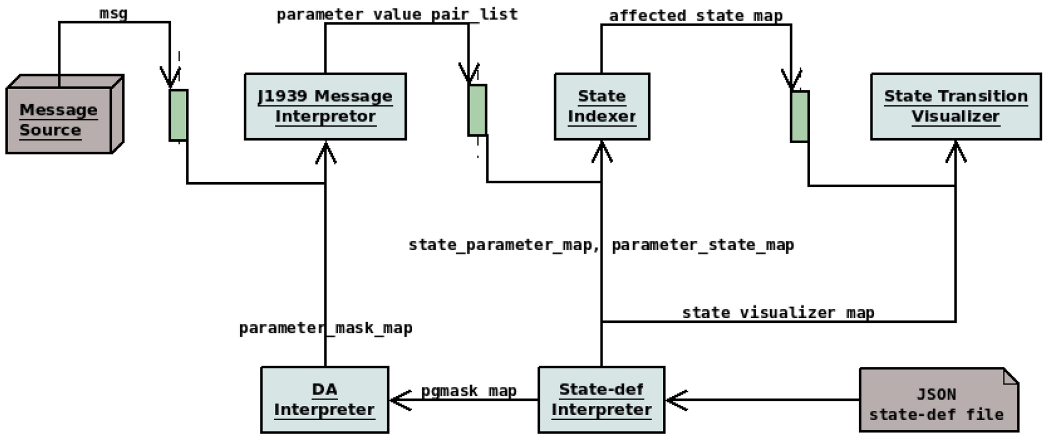


Fig. 3. TruckSTM component interaction diagram.

$\{\langle(0, 0, 11, 518), \{0.0\}\rangle\}$ 0.0% torque commanded in the SPN 518 from the PGN 0 by a malicious intruder masquerading as the brake controller ($sa = 11$).

$\{\langle(0, 0, 11, 695), \{[1.0 - 3.0]\}\rangle\}$ Engine control mode activated through the SPN 695 from the PGN 0 by a malicious intruder masquerading as the brake controller ($sa = 11$).

$\{\langle(65265, 255, 0, 84), \{[0.0 - 30.0]\}\rangle\}$ Low wheel-based vehicle speed (0.0 to 30.0 mph) broadcasted by the engine controller in the SPN 84 from the PGN 61443.

Figure 4 demonstrates an example case where user-defined states are realized from parameter values. Figure 3 demonstrates the TruckSTM components and the interactions required to achieve the goal mentioned previously. TruckSTM is not equipped with CAN transceivers and controllers for interpreting raw signals. Consequently, we allow users to enqueue such messages as a string of bits in the process input queue (which as seen in Figure 3 is also the input queue for the *J1939 Message Interpreter* (JMI)) using custom-built modules that can be easily integrated into TruckSTM. The message source can thus be anything from real or virtual CAN interfaces to even network log files. Once the messages are received in the input queue, they are delivered to JMI by a queue controller. The *queue controller* is a special process that gets messages from an input queue, passes the message to a module, waits for a module to complete processing the message, and puts the output from the module in the output queue, which is then picked up by the next queue controller. The queue controller blocks until messages are available in the queues and de-queues accordingly. It is up to the system components like JMI to make sure properly formatted messages are delivered to it. JMI converts the string of bits parameter-value pairs like the ones shown above the CAN bus in Figure 4.

Once the parameter values are obtained from a message, the next challenge is to map them to states and determine the current mode of the vehicle. We split this problem into two and designate separate modules, namely *State Indexer* (SI) and *State Transition Visualizer* (STV), to address them in parallel. Distributing the problem to two parallel modules allows us to achieve increased performance, as will be shown in Section 4.4.

We first discuss the operations performed by the SI (Section 4.2.4). The SI module essentially decides which states are affected by an incoming message. For example, at time $t1$, the incoming message with PGN 0 affects the first state $s1$ as values 0.0 and 2.0 match the corresponding state definition. This, however, does not activate the state, as $s1$ still constitutes other parameters that do not get affected. To decide if a state is affected, SI needs to map the incoming parameter value to

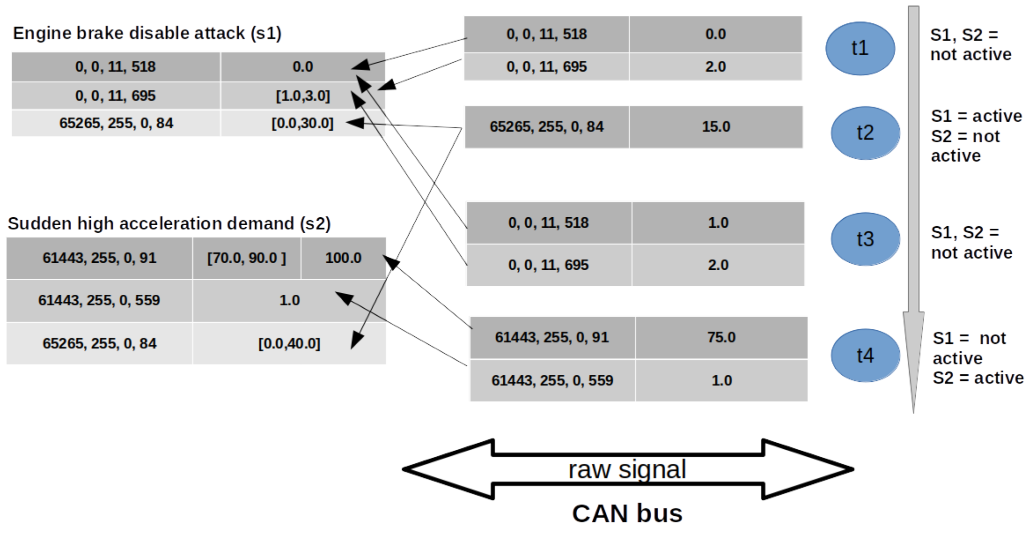


Fig. 4. Mapping parameter values to states.

the appropriate parameter instance in a state definition. To achieve this, SI maintains two indexes on the set of states. The first index is a hash map that assigns parameter instances to states that are affected by it. For example, from Figure 4, the parameter ((0, 0, 11, 518)) instance (0.0) affects $s1$, and hence this state (or a set of states) can easily be retrieved by querying the hash map, $h(0.0) = s1$. However, parameter instances can also be intervals (Section 3.3.2). For a given parameter, finding the number of interval instances that a value maps to can be a linear problem in the number of intervals. However, we can use an *interval tree* [2] that allows $O(\log n + t)$ query time (where n is the total number of intervals on which the tree is built and t is the number of intervals that are returned by the query) and returns all (including overlapping) intervals that contain a given query point. In TruckSTM, we make use of interval trees to create a second index that maps intervals to states. As an example, let us consider parameter (65265, 255, 0, 84) from Figure 4. This parameter is associated with two overlapping interval instances ([0.0 − 30.0], [0.0 − 40.0]) that affect states $s1$ and $s2$, respectively. An interval tree can thus be constructed on these interval instances. At time $t2$, when parameter (65265, 255, 0, 84) instance (15.0) is received from the bus, it can be queried on the interval tree for that parameter and both intervals will be returned. For a given parameter, both the hash map and the interval tree are mapped to the parameter in the *parameter_state_map* data structure such that $parameter\_state\_map[p] = (.., hashmap, intervaltree)$. The affected states at time $t2$ are thus $s1$ and $s2$, of which $s1$ is activated as all of its constituent parameters are activated. At time $t3$, however, the value for parameter (0, 0, 11, 518) changes. This affects $s1$ negatively and deactivates it.

SI only keeps track of affected states. Whether an affected state is active is decided by the STV. After every message is processed, SI outputs a data structure *affected_state_map* that maps the individual states to the number of parameters currently active in it. STV maintains a different data structure *state_visualizer_map* that maps all defined states to the total number of parameters in their respective definitions. If the number of parameters currently active is equal to the total number of parameters, a state is considered active, otherwise not. For example, at time $t2$, since both parameters of $s1$ are active, the state is considered to be active. For a parameter with more than one instance belonging to the same state, if a match is found with any one of the instances, the parameter is considered to be active for that state. This can be seen in Figure 4, where, at time

```
[
{
"name":{
"description": "A good name for the state", "type":"string"
},
"type":{
"enum": ["Normal","Abnormal"]
},
"definition":{
"description": "The actual definition as set of parameter instances", "type":"array", "minItems":1,
"items":{"pgn":{"type":"integer"},"da":{"type":"integer"},"sa":{"type":"integer"},
"spn":{"type":"integer"},
"pinstSet":{"type":"array",
"items":{
"oneOf":[
{"anyOf":[{"type":"array","items":{"type":"numeric"}},{"type":"numeric"}]},
{"type":"ASCII"},
{"type":"binary"}]
}
},
"comment":{"description": "A comment about the parameter instance", "type":"string"}
]
}
}
}
```

Fig. 5. State definition JSON schema.

```
[{
"name":"Engine Brake Disable Attack",
"type": "Abnormal",
"definition":
[
{"pgn":0,"da":0,"sa":11,"spn":518,"pinstSet":[0.0],"comment":"0% requested torque"},
{"pgn":0,"da":0,"sa":11,"spn":695,"pinstSet":[[1.0,3.0]],"comment":"Engine Control Activated"},
{"pgn":65265,"da":255,"sa":0,"spn":84,"pinstSet":[[0.0,30.0]],"comment":"Low vehicle speed"}
]
},
{
"name":"Sudden High Acceleration Demand",
"type": "Abnormal",
"definition":
[
{"pgn":61443,"da":255,"sa":0,"spn":91,"pinstSet":[[70.0,90.0],100.0],
"comment":"High accelerator pedal press percentage"},
{"pgn":61443,"da":255,"sa":0,"spn":559,"pinstSet":[1.0],"comment":"Kickdown active"},
{"pgn":65265,"da":255,"sa":0,"spn":84,"pinstSet":[[0.0,30.0]],"comment":"Low vehicle speed"}
]
}]
```

Fig. 6. State definition JSON example.

$t4$, parameter $(61443, 255, 0, 91)$ is activated by the incoming parameter value that falls within the range $[70.0 - 90.0]$. A state remains active until one of the constituent parameters becomes inactive. STV displays the mode of operation. In Figure 4, at time t_2, the vehicle is in mode $M_2 = \{s_2\}$ and is in mode $M_3 = \phi$ at time t_3.

4.2 Component Design and Interaction

4.2.1 State-def Interpreter. State definitions are an integral part of this project. As noted in Section 3.3, states are defined as a set of parameter instances where instances may be closed intervals or discrete values corresponding to the parameter (pgn, da, sa, spn). To simplify the syntax for state definitions, we use a simple JSON schema as shown in Figure 5. For reference, we also show a simple state definition file with two states (used in the previous section) in Figure 6.

Along with the schema, we also show two example states in the *state-definition file* from Figure 6: `High Acceleration Demand`, expressed in terms of the amount of accelerator pedal press, state of the accelerator kickdown switch, and the engine speed, whereas `Engine Brake Disable Attack` is expressed in terms of the engine requested torque and vehicle speed [1]. The schema from Figure 5 describes some of the constituent key-value pairs. For example, `name` represents a user-provided string that describes the state and `type` can be either `Normal` or `Abnormal`, thus denoting a user-specified criticality for the state. Our visualizer displays `Normal` states using the color green and `Abnormal` states using the color red (Section 4.2.5). The `definition` property presents the actual state definition as a set of `(p,pinstSet)` in accordance with Definition 1. Additionally, a comment property is also available to provide some additional information about the parameter instances. The property `pinst` is a set that can contain either ASCII elements or binary elements or a set of closed intervals of numeric elements or single-valued numeric elements.

Algorithm 1 shows the operations of the *State-def Interpreter* (SDI). This component is responsible for generating some of the most critical data structures to be used at runtime. As shown in Figure 3, it retrieves the state definition from the *state-definition file* and parses the JSON content to obtain a list of user-defined states. It then traverses through each state and populates four critical data structures:

pgmask_map: Updating the *pgmask_map* data structure begins by generating a *pgmask*. A *pgmask* is essentially a bit mask that represents the fields *pgn*, *da*, and *sa* (Figure 2) for a parameter in the state definition. The logic for obtaining these fields is shown in Section 3.2.2. The *pgmask* is generated using the reverse logic as shown in lines 13 through 19 of Algorithm 1. The *pf* is first obtained from the *pgn,* and if it is less than 240, the *pgmask* is composed using the *edp*, *dp*, *pf*, *da*, and *sa* bits. Otherwise, it is composed of the *pgn* and *sa* bits. The first 6 bits (from the left) of the *pgmask* are always kept to 0 to mask out those bits. Individual parameters are then put into a list that is mapped to the *pgmask*.

state_parameter_map: The *state_parameter_map* maps individual states to a *parameter_active _map* and the *active_state_counter*. The *parameter_active_map* maps a parameter to a bit that indicates whether that parameter is active. The *active_state_counter* denotes the number of active parameters for the state.

parameter_state_map: The *parameter_state_map* maps individual parameters to a 2-tuple. The first element of the tuple assigns a single bit to the state being processed. Setting the bit to 1 implies that the state is affected by the parameter. The second element of the tuple is a map from parameter instances to a set of states that they affect. The SI module treats the second element of the tuple as the hash map index and builds the interval tree index from intervals in its keys (refer to Section 4.1).

state_visualizer_map: The *state_visualizer_map* associates states with their *type*s obtained from the state definition file and the total number of parameters in the state that will be used later to evaluate if a state is active.

4.2.2 *DA Interpreter.*

To interpret a parameter embedded in a J1939 message, it needs to be extracted from the data field (refer to Section 3.2.2). Every SPN included in the J1939 DA is assigned to a *position* and *length* that determines the data bits that correspond to that parameter. The extracted bits can then be transformed into their appropriate data representation ($\in V_{Pn} \cup V_{Pa} \cup V_{Pb}$) using the *resolution* and *offset* factors. As seen in Figure 3, the SDI forwards the *pgmask_map*. The *DA interpreter* (DI) then generates a bit mask for every parameter using the information that it obtains from the DA. The bit mask can then be used to extract the parameter bits from the 64-bit data field. Algorithm 2 shows the process of mask generation in detail.

ALGORITHM 1: State-def Interpreter

```
Input: state definition
Output: pgmask_map, state_parameter_map, parameter_state_map, state_visualizer_map
$pos \leftarrow 0$, $active\_parameter\_counter \leftarrow 0$
$pgmask\_map \leftarrow \emptyset$                                   ▷ Data structure for J1939 Message Interpreter
$state\_parameter\_map \leftarrow \emptyset$, $parameter\_state\_map \leftarrow \emptyset$   ▷ Data structures for State Indexer startup
$state\_visualizer\_map \leftarrow \emptyset$                         ▷ Data structure for State Visualizer startup
foreach $state$ do
    $s \leftarrow (state[name], pos)$               ▷ adding the monotonically increasing pos makes s unique
    $pos \leftarrow pos + 1$
    $p_s \leftarrow$ set of parameters in this state definition
    $parameter\_active\_bit\_map \leftarrow \emptyset$
    for $p \in p_s$ do
        $parameter\_active\_bit\_map[p] \leftarrow 0$
        ▷ Update pgmask_map
        $pf \leftarrow pgn \wedge 00FF0000$
        if $pf < 240$ then
            $pgmask \leftarrow (000000\ edp_2\ dp_2\ pf_2\ da_2\ sa_2)_{10}$
        end
        else
            $pgmask \leftarrow (000000\ pgn_2\ sa_2)$
        end
        if $pgmask \notin keys(pgmask\_map)$ then
            $pgmask\_map[pgmask] \leftarrow \emptyset$
        end
        $pgmask\_map[pgmask] \leftarrow pgmask\_map[pgmask] \cup p$
        ▷ Update parameter_state_map
        if $p \notin keys(parameter\_state\_map)$ then
            $p\_tup \leftarrow (\emptyset, \emptyset)$
        end
        else
            $p\_tup \leftarrow parameter\_state\_map[p]$
        end
        $p\_tup[0][s] \leftarrow 0$
        for $pinst \in pinst'$ do
            if $pinst \notin keys(p\_tup[1])$ then
                $p\_tup[1][pinst] \leftarrow \emptyset$
            end
            $p\_tup[1][pinst] \leftarrow p\_tup[1][pinst] \cup s$
        end
        $parameter\_state\_map[p] \leftarrow p\_tup$
    end
    ▷ Update state_parameter_map
    $state\_parameter\_map[s] \leftarrow (parameter\_active\_bit\_map, active\_parameter\_counter)$
    ▷ Update state_visualizer_map
    $state\_visualizer\_map[s] \leftarrow (type, |p_s|)$
end
return pgmask_map, state_parameter_map, parameter_state_map, state_visualizer_map
```

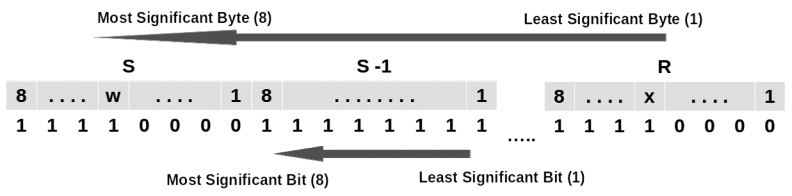
ALGORITHM 2: DA Interpreter

```
Input: $pgmask\_map$
Output: $parameter\_mask\_map$
1  $parameter\_mask\_map \leftarrow \emptyset$
2  for $pgmask \in keys(pgmask\_map)$ do
3      $parameter\_masks \leftarrow \emptyset$
4      for $p \in pgmask\_map[pgmask]$ do
5          $(position, length, resolution, offset) \leftarrow$
           $\pi_{position,length,resolution,offset}(\sigma_{PGN=pgn \wedge SPN=spn} J1939DA)$
6          parse string representations to obtain critical numbers.    ▷ refer to Section 4.2.2
7          convert length in bytes to length in bits as $length_{in\ bit} \leftarrow length_{in\ byte} * 8$
8          ▷ $position \leftarrow R.x\ r\ S.w$
9          $mask \leftarrow 00....0_2$, s.t. $|mask| = 64$ and $mask[i]$ denotes bit $i$ from the least significant side
10         $pos \leftarrow (R - 1) * 8 + x$
11         $shift\_back \leftarrow pos - 1$
12         $k \leftarrow 1$,    ▷ base case is where $length \leftarrow 1$
13         for $k \leq length$ do
14             $mask[pos] \leftarrow 1_2$
15             $k \leftarrow k + 1$
16             $pos \leftarrow pos + 1$
17             if $pos = (S - 1) * 8 + 1$ then
18                 $pos \leftarrow (S - 1) * 8 + w$
19             end
20         end
21         $parameter\_masks \leftarrow parameter\_masks \cup (p, mask, resolution, offset, shift\_back)$
22     end
23     $parameter\_mask\_map[pgmask] \leftarrow parameter\_masks$  ▷ Note that $|parameter\_masks| > 0$, always
24 end
25 return $parameter\_mask\_map$
```

As seen in lines 2 through 6 of Algorithm 2, for every parameter, the interpreter first obtains the *position*, *length*, *resolution,* and *offset* factors by querying the DA database. The obtained string representations are mostly written as semi-structured text and hence are subjected to further text processing to extract critical entities. The following four bullets describe the representation and extraction procedure for the aforementioned factors. In the process, we also make use of some formulation to describe the structure of individual textual representations:

***Length*:** Length is represented as "n u," where $n \in \mathbb{Z}$ and $u \in \{bit, bits, byte, bytes\}$. The actual length is thus the first token n. If the unit of length is in "byte" or "bytes," we convert the length into bits by multiplying by 8 as shown in line 7 of Algorithm 2.

***Position*:** Position is represented as "R.x d S.w," where $0 \leq R \leq 8$, $d \in \{\text{“}-\text{”}, \text{“},\text{”}\}$ if $d \neq \lambda$ and $R \leq S \leq 8$, $0 \leq x, w \leq 8$ when S, x, or $w \neq \lambda$, where λ denotes the empty string. Here, R and S are byte positions and x and w are bit positions in the 64-bit data field. d denotes a range operator. An example position representation from the DA can thus be "5.6 - 6.7." Two constraints hold on this representation: $((S = \lambda) \iff (d = \lambda)) \wedge ((S = \lambda) \Rightarrow (w = \lambda))$ and $(x, w = \lambda)$ indicates that the default values $x, w = 1$. The first constraint implies that if S is an empty string, then d and w are also empty. However, w can be empty irrespective of S. For example, "5.6" and "5.6 - 7" are both valid representations.



Most Significant Byte (8)
Least Significant Byte (1)
S
S -1
R
8 .... w 1 8 1
8 .... x 1
1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 ..... 1 1 1 1 0 0 0 0
Most Significant Bit (8)
Least Significant Bit (1)

Fig. 7. Parameter mask generation.

The second constraint implies that empty values for w and x mean the first bit position in a byte boundary. For example, the representation "5 - 6" is equivalent to "5.1 - 6.1." The extraction procedure therefore involves splitting "R.x d S.w" and generating two end points $R.x$ and $S.w$ or just $R.x$ in which case we set $S.w \leftarrow R.x$.

Resolution: As mentioned earlier (Section 3.2.2), resolution is generally expressed as "r unit/bit," "ASCII," and "Binary"/"l-bit bit-mapped" (l is the length of the parameter) for numerical, ASCII, and binary parameters, respectively. For numerical parameters, only the first token denoted by r is extracted. The rest ("ASCII," "Binary," "Bit-mapped") are kept unprocessed.

Offset: For numerical parameters, offset is generally expressed as "o unit," where o is either 0 or a negative number and is not used for ASCII or binary parameters. We therefore simply extract the first token (i.e., o) for usage in message interpretation of numeric parameters.

Having successfully extracted the *position* and *length* factors, we now demonstrate the process of mask generation in lines 8 through 21 of Algorithm 2. The purpose of the mask is to extract the parameter bits from the 64 (or less)-bit data field. Accordingly, we generate a 64-bit mask by setting only those bits to 1 that correspond to the *position*s of the parameter in the data field and later (in Section 4.2.3) bitwise AND this mask with the data bits to obtain the parameter bits. Let the parameter position obtained from the DA be *R.x* - *S.w*. We use the parameter placement specification from the SAE J1939 application layer documentation [15] to set bits of an all-0 64-bit mask as shown in Figure 7. We start by setting the x^{th} bit of the R^{th} byte (i.e., bit $((R-1)*8+x)$ from LSB) and incrementally set every bit until the final (8^{th}) bit of the $(S-1)^{th}$ byte. The total number of bits thus set is $(S-1)*8-((R-1)*8+x)+1$. The remaining $l-((S-1)*8-((R*8-1)+x)+1)$ bits are set incrementally starting from the w^{th} bit of byte S. The eventual 64-bit mask resembles the bit pattern 0...1111000011111.....1111000...0. Once the parameter bits are masked out by ANDing with the data fields, the result needs to bit shifted back (right) by $(R*8-1)+x-1$ bit before being interpreted.

4.2.3 *J1939 Message Interpreter.*

The JMI is the first in the chain of three components that participate in the process of realizing operational states at runtime. This component plays two critical roles. First, it filters messages which do not satisfy any state definitions and it does this in constant time. This can be seen in lines 2–4 of Algorithm 3 where it only processes those messages whose *pgmask*s match the ones received from SDI. Second, it transforms 64 bit J1939 *Data* into a list of parameter-value pairs which it then forwards to SI.

The conversion process shown in line 5–17 of Algorithm 3 is basically an algorithmic representation of the conversion logic shown in Section 3.2.2. The parameter bits are first extracted by masking with the parameter mask $pm[1]$ received from SDI. Then, depending on the data type i.e numeric, ASCII or binary the appropriate resolution and offset values are used to produce the final output. After all the parameters in a message have been processed, the list of parameter-value pairs is forwarded to SI for further processing.

ALGORITHM 3: J1939 Message Interpreter

```
Input: parameter_mask_map, msg
Output: parameter_value_pair_list
1  msg contains $ID$ and $Data$
2  $pgmask \leftarrow ID \wedge 03FFFFFF_{16}$
3  $parameter\_value\_pair\_list \leftarrow \emptyset$
4  for $pm \in parameter\_mask\_map[pgmask]$ do
5      $raw\_parameter\_value_2 \leftarrow (Data \mathbin{\&} pm[1]) \gg pm[4]$    ▷ $pm[1]$ : parameter_mask, $pm[4]$ : shift_back
6      $value \leftarrow NULL$
7      if $pm[2]$ = "ASCII" then
8          $value \leftarrow raw\_parameter\_value_8$
9          reverse $value$    ▷ ASCII data is transmitted in reverse byte order
10     end
11     else if ($pm[2]$ = "Binary") ∨ ($pm[2]$ = "Bit-mapped") then
12         $value \leftarrow raw\_parameter\_value_{10}$
13     end
14     else
15         $value \leftarrow raw\_parameter\_value_{10} * pm[2] + pm[3]$
16     end
17     $parameter\_value\_pair\_list \leftarrow parameter\_value\_pair\_list \cup (pm[0], value)$
18 end
19 return $parameter\_value\_pair\_list$
```

4.2.4 *State Indexer.* The SI's role in TruckSTM is to determine all states affected by the parameters in the input list of parameter-value pairs. To do this, it first forms two indexes at startup. While the SDI sent *parameter_state_map* contains the hash map–based index, the interval tree–based index is generated and added to the *parameter_state_map* in lines 2 and 3 of Algorithm 4.

At runtime, SI traverses through the list of parameter-value pairs forwarded by JMI, and for each parameter value it attempts to obtain to set of affected states by querying the two indexes (lines 7 through 16). It only queries the interval tree if it contains at least one interval node. For each successful query, it traverses the list of states obtained and sets the corresponding bit in the *parameter_state_map* to denote that the state has been affected by that parameter.

Until this point, the algorithm only finds those states that have been positively affected by a parameter value. However, there may be other states affected previously by a different value for that same parameter. When the value changes, these states should get affected negatively—that is, they should move closer to being deactivated. Accordingly, we traverse all states mapped to the parameter in lines 17 through 30. If we find that this state was previously positively affected by this parameter and is now negatively affected or vice versa, we alter the bit (lines 20 through 25). Finally, if the altered bit is a 0, we decrease the *active_parameter_counter* by 1 or else we increase it by 1. When a state is affected, SI puts it in a separate data structure (*active_state_map*) to be processed by STV. This data structure also stores the previously affected states, thus allowing STV to switch any previously active (transitional) states to inactive. By sending only the previously and currently affected states, SI effectively reduces the load on STV.

4.2.5 *State Transition Visualizer.* The STV serves two purposes. First, it determines the current mode of operation by classifying affected states into active, inactive, and transitional. It does so by traversing the *active_state_map* received from SI and comparing the state's

ALGORITHM 4: State Indexer

```
Input: $state\_parameter\_map$, $parameter\_state\_map$, $parameter\_value\_pair\_list$
Output: $affected\_state\_map$
▷ At startup
$active\_state\_map \leftarrow \emptyset$
$parameter\_state\_map[p][2] \leftarrow$ construct_interval_tree($intervals \in keys(parameter\_state\_map[p][1])$)
▷ At runtime
$current\_state\_map \leftarrow \emptyset$
for $(p, v) \in parameter\_value\_pair\_list$ do
    for $s \in parameter\_state\_map[p][1][v]$ do
        $parameter\_state\_map[p][0][s] \leftarrow 1$
    end
    if $parameter\_state\_map[p][2]|$ has nodes then
        for $interval \in search(parameter\_state\_map[p][2], v)$ do
            for $s \in parameter\_state\_map[p][1][interval]$ do
                $parameter\_state\_map[p][0][s] \leftarrow 1$
            end
        end
    end
    for $s \in keys(parameter\_state\_map[p][0])$ do
        if $state\_parameter\_map[s][0][p] \neq parameter\_state\_map[p][0][s]$ then
            $state\_parameter\_map[s][0][p] \leftarrow parameter\_state\_map[p][0][s]$
            if $state\_parameter\_map[s][0][p] = 0$ then
                $state\_parameter\_map[s][1] \leftarrow state\_parameter\_map[s][1] - 1$
            end
            else
                $state\_parameter\_map[s][1] \leftarrow state\_parameter\_map[s][1] + 1$
            end
            $active\_state\_map[s] \leftarrow state\_parameter\_map[s]$
            $current\_state\_map[s] \leftarrow state\_parameter\_map[s]$
        end
        $parameter\_state\_map[p][0][s] \leftarrow 0$
    end
end
$current\_previous\_affected \leftarrow active\_state\_map$
$active\_state\_map \leftarrow current\_state\_map$
return $current\_previous\_affected$
```

active_parameter_counter with the total parameter counter received from the *state_visualizer_map*. This process is shown in Figure 8. If the *active_parameter_counter* is equal to the total parameter counter, a state is active; otherwise, it is inactive or transitional depending on whether it was transitional or active previously. An additional pop-up text, which denotes the currently active and inactive constituent parameters, is also provided on the *visual_object*. Clicking on a state button displays the pop-up form.

As an example, Figure 9 shows three instances of STV's display window at runtime. The states exhibited in this example follow the same definition and activation timing as seen in Figures 4 and 6. Initially, both states are off (grey). At time *t2,* state 1 ("Engine brake disable attack")

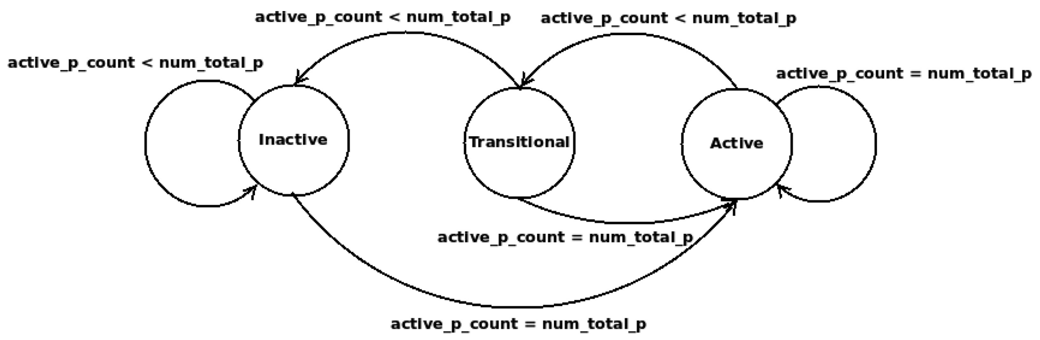

active_p_count < num_total_p
active_p_count < num_total_p
active_p_count < num_total_p
active_p_count = num_total_p
Inactive
Transitional
Active
active_p_count = num_total_p
active_p_count = num_total_p

Fig. 8. State visualizer.

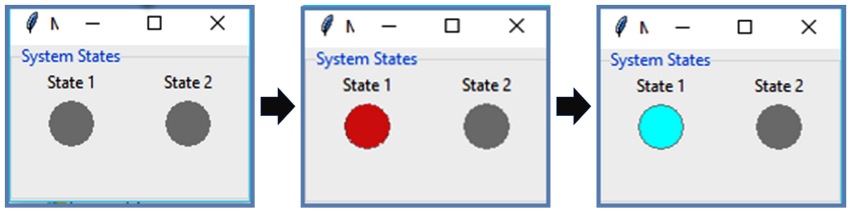


Fig. 9. State visualizer window.

becomes active (`red` denoting "abnormal" state), and at *t3,* state 1 switches off again, thus showing a transition from mode M_2 to mode M_3.

The algorithms demonstrate that they can perform high-level state visualization in real time. We verify the correctness of our algorithms by validating them against test cases and observing their behavior. We provide no soundness or completeness proof because we experimented with only a few states to demonstrate the feasibility of the approach.

4.3 Implementation, Extensibility, and Adaptability

TruckSTM is implemented in Python and can be run on different platforms. Individual modules execute as separate processes communicating using the FIFO data structures. For this, we use publicly available libraries. However, these libraries can have inherent processing delays that sometimes can vary as a function of the input. Accordingly, processing time can be dependent on the underlying library's performance. TruckSTM currently includes commercial and standard information like the DA and some recorded logs. The code will be provided to an interested party if requested.

TruckSTM, however, is designed modularly—that is, individual modules can be modified or replaced without requiring to make extensive changes to other modules. This means that TruckSTM can be extended in the future to provide enhanced functionality. TruckSTM can also be easily integrated with other Python modules. For example, a researcher willing to plot J1939 interpreted messages can read from the corresponding output queue and plot signal values.

TruckSTM can be adapted to other systems and networks, as it is not dependent on the underlying networking hardware's capability. Furthermore, modifying the JMI and SDI logic to suit other application layer standards (like those seen in passenger cars) can result in the same functionality. The state definition format can be used in other domains, possibly with different parameter identifiers.

4.4 Performance Analysis

In this section, we will attempt to analyze TruckSTM's asymptotic performance in processing a single message. Because TruckSTM modules are piped, the time required to process a single message is dependent on the maximum of the time required by any of the three modules working in parallel. In addition, TruckSTM has two input sources, namely the state definition file and message source. Thus, TruckSTM's performance can be affected by the characteristics of a message and that of the state definition file. However, factors such as bus load or message frequency do not affect performance, as incoming messages are always stored in a queue and filtered significantly by JMI.

Let us assume that n_s = the number of states in a arbitrary state definition file, $n_{p_{msg}}$ = the number of parameters in an arbitrary received message, and n_{pinst} = the number of parameter instances (intervals and discrete values) that correspond to an arbitrary parameter in the message. Let the number of states affected by a parameter be n_{s_p} and the number of states affected by an arbitrary parameter instance be $n_{s_{pinst}}$. Finally, let the number of parameter instances matching a given value be n_{pinst_v} and the number of states affected after a message is processed be $n_{s_{msg}}$. According to the J1939 standards [9], $n_{p_{msg}}$ is upper bounded at a value of 32. Consequently, we assume it to be equal to a non-varying quantity c.

Asymptotic time complexity for JMI. As seen from Algorithm 3, JMI loops through the number of parameters in a message. Since $n_{p_{msg}} = c$, its time complexity is constant (i.e., $O(1)$).

Asymptotic time complexity for SI. From Algorithm 4, we calculate SI's time complexity in a line-by-line fashion to demonstrate the detailed calculations:

Lines 7 through 9: $O(n_{s_{pinst}})$
Line 11: Search operation on interval tree: $O(log(n_{pinst}) + n_{pinst_v})$
Lines 11 through 15: $O(\sum_1^{n_{pinst_v}} n_{s_{pinst}})$
Lines 17 through 30: $O(n_{s_p})$
Lines 6 through 31: $O(n_{s_{pinst}}) + log(n_{pinst}) + n_{pinst_v} + \sum_1^{n_{pinst_v}} n_{s_{pinst}} + n_{s_p})$, neglecting the total number of $n_{p_{msg}}$ or c loops.

In the best case, if no state is positively affected and the interval tree is an empty leaf (i.e., $n_{s_{pinst}} = 0$ and $n_{pinst} = 1$), complexity is still $O(n_{s_p})$. This is because states may be negatively affected, even if no state has been positively affected. For negatively affected states, the concerned parameter will have to be deactivated. In the worst case, $n_{pinst_v} = n_{pinst}, n_{s_{pinst}}, n_{s_p} = n_s$, implying a worst-case time complexity of $O(n_s * n_{pinst})$. This means that when all parameter instances are intervals and correspond to all states in a definition, the performance of SI decreases rapidly as the number of states and the number of parameter instances increase.

Asymptotic time complexity for STV. STV only traverses the list of states returned by SI. Let this number be denoted as $n_{s_{aff}}$. Because affected states from two consecutive messages at time instances t_i and t_j can overlap, $n_{s_{aff}} \leq n_{s_{msgt_i}} + n_{s_{msgt_j}}$. The asymptotic time complexity of STV is this $O(n_{s_{aff}})$. In the best case, if no state is affected at time t_j, STV still traverses all $n_{s_{msgt_i}}$ messages. In the worst case, $n_{s_{aff}} = n_s$, implying a worst-case time complexity of $O(n_s)$. This means that in the worst case when all parameters affect all states in a definition, the performance of STV decreases as fast as the number of states increases.

5 PERFORMANCE TESTING

We now focus on the testing of TruckSTM's performance when subjected to worst-case conditions so that we can set upper bounds on our application's performance.

```
00FCB400 # 0000000000000000
00FCB400 # FFFFFFFFFFFFFFFF
00FCB400 # 0000000000000000
00FCB400 # FFFFFFFFFFFFFFFF
00FCB400 # 0000000000000000
00FCB400 # FFFFFFFFFFFFFFFF
00FCB400 # 0000000000000000
00FCB400 # FFFFFFFFFFFFFFFF
00FCB400 # 0000000000000000
00FCB400 # FFFFFFFFFFFFFFFF
```

(a) Message Stream

```
{
"name":"T1",...
[
{"pgn":64692,"da":255,"sa":0,"spn":6048,"pinst":[[-0.1,0.1],[-0.2,0.2],...],
"comment":""}
{"pgn":64692,"da":255,"sa":0,"spn":6049,"pinst":[[-0.1,0.1],[-0.2,0.2],...],
"comment":""}
.
.
]
},
{name:"T2", ....{"pgn":64692,"da":255,"sa":0,"spn":6048,"pinst":.....],
"comment":""}},
{name:"T2", ....{"pgn":64692,"da":255,"sa":0,"spn":6048,"pinst":.....],
"comment":""}},
]
```

(b) State Definition File

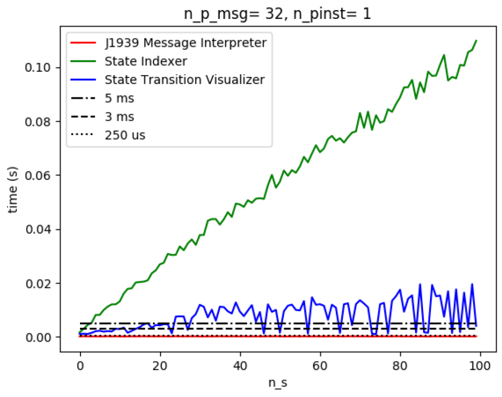
Fig. 10. Worst-case experiment inputs.

5.1 Worst-Case Analysis

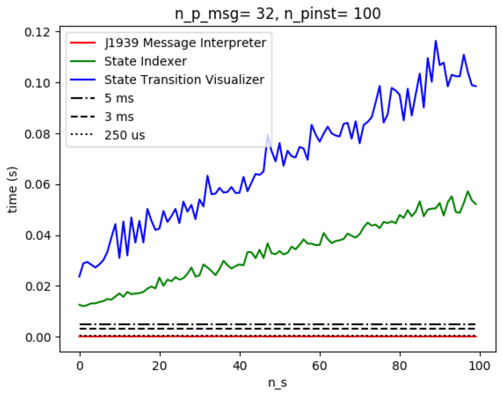
5.1.1 Experiment Setup. As per Section 4.4, the performance of SI and STV is affected by the number of states in a state definition file n_s and the maximum number of parameter instances for all parameters in a message, n_{pinst}. Given this, we decided to conduct our experiments varying the n_s and n_{pinst} values and keeping the number of parameters per message fixed at the maximum 32. TruckSTM relies on external modules that may slow down the process if it becomes complex. Consequently, we decided to vary the experiment independent factors n_s and n_{pinst} within the range of 1 to 100. PGN 64692 is associated with 32 different SPNs and hence could form 32 different parameters.

TruckSTM receives data from the state definition file and the message source. We need to manipulate both of these sources to demonstrate worst-case scenarios. The worst-case performance is achieved if all parameters in a state definition file are included in all states and all parameter instances are parts of all states. This can be seen in Figure 10(b), where all 32 parameters from PGN 64692 are included in all states. Furthermore, all parameters are assigned to the same number of instances such that every instance includes the value 0.0. This is done to ensure that if all parameter values are set to 0, all parameter instances match, thereby affecting all states. The message stream used for this purpose is shown in Figure 10(a). A sequence of 10 alternating messages is transmitted onto a virtual CAN interface in rapid succession. The alternating sequence allows all states to be affected for every single message. If a sequence of same messages is used, all states will be activated for the message and never deactivated. Using this technique, every module participates in processing every message, and that switches states on and off. For every experiment, a new state definition file is generated by varying the number of states and the number of parameter instances. We conducted our experiments on a 64-bit Fedora Linux machine with two quad-core Intel Xeon @2.6 GHz processors. Messages were read from the virtual CAN interface and replayed onto the input queue.

5.1.2 Analysis and Discussion. From Section 4.4, it was shown that JMI's worst-case performance is constant in time, SI's worst-case performance was linear with both n_s and n_{pinst}, and STV's worst-case performance depended linearly only on n_s. These characteristics can be clearly observed in Figures 11 and 12. The red curve represents the performance characteristics on JMI, which is not affected by increasing n_s or n_{pinst} values. Next, the green curve shows the characteristic of SI. This curve is linear with respect to both n_s and n_{pinst}. Finally, the blue curve shows the performance of the visualizer, which although considerably more fluctuating is unaffected by

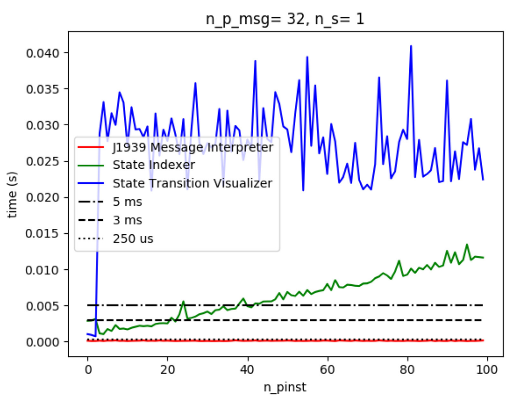


(a) Varying n_s, with minimum n_{pinst}

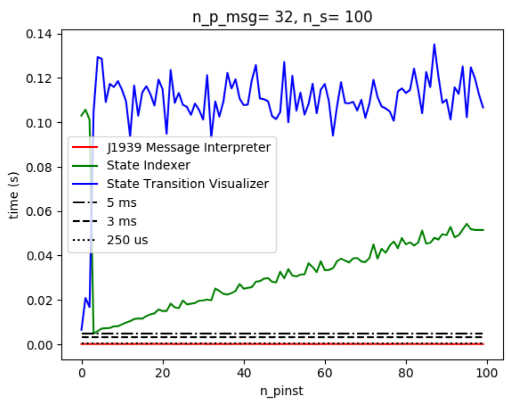


(b) Varying n_s, with maximum n_{pinst}

Fig. 11. TruckSTM worst-case performance curves for increasing n_s.



(a) Varying n_{pinst}, with minimum n_s



(b) Varying n_{pinst}, with maximum n_s

Fig. 12. TruckSTM worst-case performance curves for increasing n_{pinst}.

increasing n_{pinst} values (Figure 12) but linearly affected by the number of states. In general, STV consumes more time than SI. This is because STV is responsible for painting and repainting visual objects, which is a fairly resource-intensive operation even on most modern-day computers. The other important observation from both the figures is that in most cases only JMI performs within both the hard and soft timing constraints introduced in Section 1. The other modules are generally over both constraints.

Note that we do not evaluate the runtime complexity of the queue controller (Section 4.1) because enqueue and dequeue operations are constant time in theory. Implementers must make sure that the near to constant time, complexity is maintained to avoid clogged queues and performance bottlenecks. One way to achieve this would be to implement custom queue controllers that work on a publish-subscribe basis. In such a model, an incoming queue controller dequeues a message only when its corresponding TruckSTM component is done processing the previous message and enqueues the output (if available) only when the current message is done processing.

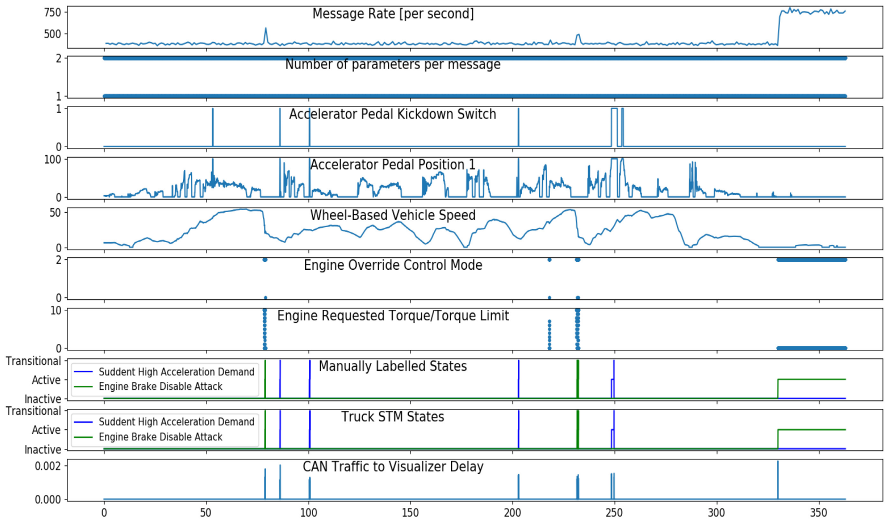


Fig. 13. Real-world performance analysis.

5.2 Real-World Application Analysis

The worst-case analysis shown earlier is highly pessimistic with respect to the real-world application of TruckSTM. First, these tests are performed under the worst-case test scenario. This is practically impossible, as replicating the same state definitions is unrealistic and will only hamper system performance. Second, it is highly unlikely that the same parameter instance or even the same parameter will be present in and/or affect all states. This will not only reduce the load on STV but also require SI to perform a lesser number of traversals. Finally, it is highly unlikely that all parameter instances will be intervals and an incoming value will match all. As an example, close to 50% of the SPNs defined in the J1939 DA are switch states that assume only a very small set of values. Thus, in most cases, instances for switch states will not be expressed as intervals. This is also true for ASCII and binary parameters. In most cases, the parameter will not have overlapping intervals. This will reduce the number of intervals returned by the interval tree and hence speed up SI's performance significantly. Thus, when used in a real-world scenario, we expect TruckSTM to perform much faster than what has been shown in Figures 11 and 12. To validate this statement, we perform a real-world (average case) analysis in this section.

5.2.1 Experiment Setup. We have simulated a scenario by using a previously collected CAN trace [3] from a Paccar MX–powered Kenworth tractor driven around a parking lot. The trace reflects approximately 6 minutes of driving activity including three hard-braking scenarios during which PGN 0 was transmitted naturally by the ABS. This same trace was used to generate the TruckSTM snapshots from Figure 9. The recorded trace was then replayed on a virtual CAN network with the same message inter-arrival delays. This simulated a real-world driving scenario, although a remote telematics server would typically experience additional network communication delays. Figure 13 shows the critical bus statistics and parameter values in the first seven rows from the top. The state definitions from Figure 6 are used in this case. To better visualize the "Engine Brake Disable Attack," we manually inserted PGN 0 to override engine control toward the end of the trace (from 330 seconds onward). The message rate doubled as a result of the injection.

Based on parameter values obtained from the trace, we simulated the ideal scenario by manually tagging the states using the theoretical descriptions from Section 3. The manually tagged states (shown in row 8) are placed at the exact time instances when the constituent parameters attained the triggering values. When TruckSTM is run on the replayed traffic, states are visualized and shown in row 9 from the top along with the delays (in seconds) in reflecting the transitions in the bottom-most row.

5.2.2 Analysis and Discussion. There are two critical observations from Figure 13. First, all manually labeled states/modes and transitions are displayed correctly by TruckSTM. Second, the time delay is significantly low (less than 5 ms) and does not vary with the message rate or the number of parameters in a message that fluctuates between one and two throughout the replay. Both the hard and soft timing constraints are met in the real-world scenario. It must, however, be noted that the timing constraints may not be the same for all modules in TruckSTM. This is because the message filtering and state indexing functions performed by the SI and STV modules are not required to function as fast, as messages may be sent on the network. For example, if we assume that in 10% of the messages/PGNs on the network are used in state definitions, SI gets a 10 times larger window for processing every message. Moreover, not all interpreted messages affect states. If we assume that only 10% of the incoming parameter values affect states, STV gets a 100 times larger window for processing every message, thereby relaxing its hard timing constraint significantly.

Another observation from Figure 13 is that the "Engine Brake Disable Attack" is enabled on some occasions before the 330-second mark even though it was not performed explicitly. This necessitates that state definitions are chosen correctly or other criteria, like period of activity of a state, be used to infer some high-level information from TruckSTM outputs.

6 CONCLUSION AND FUTURE WORK

In this work, we presented TruckSTM, a visual system that allows users to configure high-level states like braking, accelerating, driving, cranking, or even complicated security states using vehicle parameters that can be obtained from in-vehicle traffic. Most medium and heavy duty vehicles support the SAE J1939 standards[5] that ensure interoperability across multiple vendors. This work thus has wide applicability. We developed a prototype that demonstrates the feasibility of our approach and demonstrated that the performance constraints are satisfied in general. The design of TruckSTM is modular and hence can easily be extended or adapted to suit other systems and networks like passenger cars.

In the future, we plan to improve the performance and the user interface. We plan to enhance STV to display the mode and state transitions in a more intuitive manner using visible edges instead of color combinations. We plan to integrate TruckSTM into a Web-based system so that it can be used by fleet managers, vehicle vendors, and independent researchers. We plan to verify correctness of the TruckSTM states in the future using knowledge from automotive engineers. We also plan to adapt TruckSTM so that it can read multi-packet messages.

ACKNOWLEDGMENTS

We thank Dr. Jeremy Daily (University of Tulsa) and Dr. Urban Johnson (National Motor Freight Traffic Association) for guiding us on various aspects of this work.

REFERENCES

[1] Y. Burakova, B. Hass, L. Millar, and A. Weimerskirch. 2016. Truck hacking: An experimental analysis of the SAE J1939 standard. In *Proceedings of the 10th USENIX Workshop on Offensive Technologies (WOOT'16)*. 211–220.

[5]Although other proprietary protocols and some standard ones like Universal Diagnostic Services (UDS) maybe used, they are mostly used for purposes that do not directly influence the runtime behavior of the vehicle.

[2] H. Edelsbrunner. 1980. *Dynamic Data Structures for Orthogonal Intersection Queries*. Technische Universität Graz/Forschungszentrum Graz. Institut für Informationsverarbeitung.
[3] S. Mukherjee, J. Walkery, I. Rayz, and J. Daily. 2017. A precedence graph-based approach to detect message injection attacks in J1939 based networks. In *Proceedings of the 15th Annual Conference on Privacy, Security, and Trust (PST'17)*. IEEE, Los Alamitos, CA.
[4] S. N. Narayanan, S. Mittal, and A. Joshi. 2016. OBD_SecureAlert: An anomaly detection system for vehicles. In *Proceedings of the 2016 International Conference on Smart Computing (SMARTCOMP'16)*. IEEE, Los Alamitos, CA, 1–6.
[5] W. Quanqi, W. Jian, and W. Yanyan. 2011. Design of vehicle bus data acquisition and fault diagnosis system. In *Proceedings of the International Conference on Consumer Electronics, Communications, and Networks (CECNet'11)*. IEEE, Los Alamitos, CA, 245–248.
[6] R. Bosch. 1991. *CAN Specification Version 2.0*. Robert Bosch GmbH.
[7] H. Said, T. Nicoletti, and P. Perez-Hernandez. 2016. Utilizing telematics data to support effective equipment fleet-management decisions: Utilization rate and hazard functions. *Journal of Computing in Civil Engineering* 30, 1 (2016), 04014122.
[8] W. Sun, J. Li, Y. Gao, D. Qu, and C. Yang. 2010. The design of embedded bus monitoring and fault diagnosis system based on protocol SAE J1939. In *Proceedings of the Asia-Pacific Power and Energy Engineering Conference*. IEEE, Los Alamitos, CA, 1–4.
[9] Truck Bus Control and Communications Network Committee. 2015. J1939 Digital Annex. Retrieved October 7, 2019 from http://standards.sae.org/j1939da_201510/.
[10] Truck Bus Control and Communications Network Committee. 2000. *Recommended Practice for Control and Communications Network for On-Highway Equipment, SAE J1939-01*. SAE International. http://standards.sae.org/j1939/1_200009/.
[11] Truck Bus Control and Communications Network Committee. 2009. *Surface Vehicle Recommended Practice, SAE J1939*. SAE International. http://standards.sae.org/j1939_200903/.
[12] Truck Bus Control and Communications Network Committee. 2014. *Network Layer, SAE J1939-31*. SAE International. http://standards.sae.org/j1939/31_201404/.
[13] Truck Bus Control and Communications Network Committee. 2015. *Data Link Layer, SAE J1939-21*. SAE International. http://standards.sae.org/j1939/21_201504/.
[14] Truck Bus Control and Communications Network Committee. 2015. *Reduced Physical Layer, 250K bits/sec, UN-Shielded Twisted Pair (UTP), SAE J1939-15*. SAE International. http://standards.sae.org/j1939/15_200808/.
[15] Truck Bus Control and Communications Network Committee. 2015. *Vehicle Application Layer, SAE J1939-71*. SAE International. http://standards.sae.org/j1939/71_201506/.
[16] Truck Bus Control and Communications Network Committee. 2016. *Physical Layer, 250K bits/s, Twisted Shielded Pair, SAE J1939-11*. SAE International. http://standards.sae.org/j1939/11_200609/.
[17] Vector Informatik. 2016. CANoe ProductInformation. Retrieved October 7, 2019 from https://vector.com/portal/medien/cmc/info/CANoe_ProductInformation_EN.pdf.
[18] M. Wolf and R. Lambert. 2017. Hacking trucks—Cybersecurity risks and effective cybersecurity protection for heavy duty vehicles. In *Automotive—Safety & Security 2017—Sicherheit und Zuverlassigkeit fur automobile Informationstechnik*, P. Dencker, H. Klenk, H. B. Keller, and E. Plodererder (Eds.). Gesellschaft fur Informatik, Bonn, Germany, 45–60.
[19] M. Zhang, C. Chen, T. Wo, T. Xie, M. Z. Bhuiyan, and X. Lin. 2017. SafeDrive: Online driving anomaly detection from large-scale vehicle data. *IEEE Transactions on Industrial Informatics* 13, 4 (2017), 2087–2096.
[20] T. Zhu, Z. Wei, Y. Wu, and R. Zhai. 2016. The parameter analysis system of CAN bus for electric vehicle based on LabVIEW. In *Proceedings of the 6th International Conference on Machinery, Materials, Environment, Biotechnology, and Computer*. 1362–1366.

Received December 2017; revised October 2018; accepted December 2018