VeRV: A Temporal and Data-Concerned Verification Framework for the Vehicle Bus Systems

Shuo Zhang, Fei He and Ming Gu
Tsinghua National Laboratory for Information Science and Technology (TNList)
School of Software, Tsinghua University
Key Laboratory for Information System Security, Ministry of Education, China
zhangshuo12@mails.tsinghua.edu.cn, {hefei, guming}@tsinghua.edu.cn

***Abstract*—As a part of the international standard IEC 61375, the multifunction vehicle bus (MVB) has been used in most of the modern train control systems. It is highly desirable to check the temporal properties of the data transmitted on the bus. However, we are not aware of any published work on this problem. We proposed *VeRV*, the first temporal and data-concerned verification framework for the vehicle bus systems. A domain-specific language, called *VeSpec*, is proposed to specify the packet formats and the desired properties. The language is expressive, modular and easy to use. Given a *VeSpec* script, the *VeRV* allows automatic generation of runtime analyzer. We have applied our technique to a real tube train system and succeeded in diagnosing a real failure in this system. The industry application illustrates the effectiveness and efficiency of our technique.**

Keywords*—Vehicle bus systems, runtime verification, domain-specific language, onling monitoring

I. Introduction

The train communication network (TCN) was adopted in 1999 as the international standard (IEC 61375) [1], [2]. As a part of the TCN, the multifunction vehicle bus (MVB) interconnects on-board equipment in a vehicle. The MVB is used in most of the modern train control systems.

The transmitted data in an MVB may contain information of the connected equipment. For example, Fig. 1 (a) shows a packet conveying values of three variables: v_{speed} standing for the speed of the vehicle, v_{temp} representing the temperature inside the vehicle, and b_{ac} indicating if the air condition in the vehicle is turned on. One may want the following property to hold: the air condition is turned on if the temperature is above 27°C, and turned off if the temperature is below 23°C, which can be specified in temporal logic as:

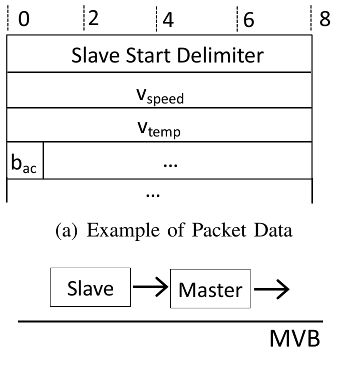
$$[\,]\,((v_{temp} > 27 \ \wedge \ b_{ac}) \vee (v_{temp} < 23 \ \wedge \ !b_{ac})),$$

where [ ] is the "always" temporal operator.

Fig. 1 (b) shows that two types of packets are transmitted on the MVB, i.e., the master packets and the slave packets. It is required that a slave packet must be preceded by a master packet. In temporal logic, this property is expressed as:

$$[\,]\,(\mathit{IsSlave} \rightarrow (*)\,\mathit{IsMaster}),$$

where *IsSlave* and *IsMaster* are two predicates indicating that the packet is a slave (resp. master) frame, and (*) is the "in the previous" temporal operator. Note that this property involves two sequential packets transmitted on the bus.



(a) Example of Packet Data

(b) Packet Sequence

Fig. 1: Packets Transmitted On the MVB

The third example lies in the failure prediction of the train control system. Consider that many sensors are connected by the MVB to a central controller in the vehicle, these sensors are responsible to periodically report their detected data to the controller. If certain data (for example the temperature data) has not been reported for a long time, we can infer that the corresponding sensor (i.e., the temperature sensor) may has gotten wrong.

For all of the three above examples, a monitor or a runtime analyzer is highly desired, with which we can record the values of related variables and check the above data-concerned[1] properties at runtime. Note that in many complicated situations, these properties involve not only the data conveyed in the current packet, but also those conveyed in previously received packets, i.e. they are temporal properties.

We are unaware of any work that checks the temporal and data-concerned properties for the MVB systems. There are a wide variety of works in network analysis [3], [4]. However, these techniques are mostly applicable to the Internet. And these techniques mainly concern the communication protocol but not the transmitted data. Thus, these techniques cannot be directly applied to our problem. On the other side, there are some commercial products [5]–[8] for analyzing the MVB. However, all these products focus on testing or simulation of the MVB. They cannot handle temporal properties at runtime.

[1]By data-concerned, we mean the data conveyed in the packets are of concern.

According to the standard [1], the MVB employs the real-time protocols (RTP) for data communication. Two classes of data are transmitted in the MVB: the time-critical and short *process data*, and the less urgent but perhaps lengthy *message data*. We focus on the process data which is more critical for the train control system. From the standard we have learned that the transmission of process data has following characteristics:

1) the process data is broadcast to all devices on the same bus;
2) the process data is transmitted cyclically at a fixed period;
3) the format of packets (including the start position and occupied length of each process data) is statically configured before operation.

With these characteristics, extracting process data automatically from the packets is possible.

We present $VeRV$, a temporal and data-concerned verification framework for the MVB, in this paper. It combines formal method and network analysis to verify temporal properties at runtime. In general, $VeRV$ involves extracting process data from the packets, recognizing events from their value changes and checking properties by a verification engine. $VeRV$ allows automatic generation of runtime analyzer. It uses the *VeSpec* language to describe the packet formats and specify the desired properties. *VeSpec* employs the past time linear temporal logic ($ptLTL$) [9] [10] to specify the properties. The language is expressive, modular and easy-to-use. One can express properties in a modular way.

The main technical contributions of this paper are summarized as follows:

- We present the first temporal and data-concerned analyzer for the MVB.
- We present a domain-specific language for specifying packet formats and temporal properties of the MVB.
- We have applied our technique to a real tube train system and succeeded in diagnosing a real failure in this system.

The rest of the paper is organized as follows. We first give a brief introduction to multifunction vehicle bus in section II. Then some preliminaries and definitions are presented in section III. The main work flow of $VeRV$ is elaborated in section IV. And the *VeSpec* language is introduced in section V. Section VI shows our experiments on a real tube train system and illustrates the effectiveness of $VeRV$. Finally, we introduce related works in section VII and conclude this paper in section VIII.

II. Multifunction Vehicle Bus

In this section, we briefly introduce the multifunction vehicle bus (MVB) which is a part of the train communication network [1].

According to the standard [1], a TCN is composed of two buses, a multifunction vehicle bus (MVB) and a wire train bus (WTB), where the MVB works for interconnecting the on-board equipment in one vehicle, and the WTB is responsable for interconnecting the many vehicles in a train. Fig. 2 shows a typical TCN system, where two vehicles are connected by a WTB and the devices in each vehicle is interconnected by an MVB. The devices connected by an MVB include the brake subsystem, the central controller, the air conditions, the lights, alarms, etc.

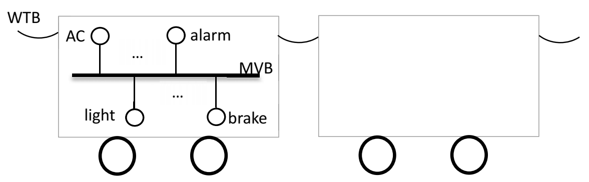
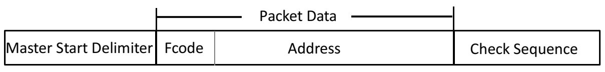
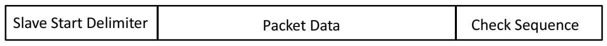


Fig. 2: The multifunction vehicle bus



(a) Master Packet



(b) Slave Packet

Fig. 3: Packet Formats in the MVB

Two kinds of devices are recognized by an MVB: the master device and the slave devices. On an MVB, there is one and only one master device, and there are number of slave devices. The master device coordinates the communication on the MVB by sending the master packet periodically. A master packet contains requests for some data, and is broadcasted to all devices on the bus. The (only) slave device which maintains the requested data responses by a slave packet. This slave packet contains the current value of the requested data. It is also broadcasted to all devices on the bus, but only devices that subscribes to this data will receive it.

Fig. 3 show the formats of these two kinds of MVB packets. Both of them consist of three parts: the start delimiter, the packet data and the check sequence. The start delimiter indicates the start of a particular type of packet. The check sequence is used to check if there is any error during the transmission. The packet data in a master packet consists of two parts: the *Fcode* field that specifies the functionality of this packet and the *Address* field which tells the identifier of the requested data. If the value of the *Fcode* is between 0 and 4, this packet is used for requesting a process data. The packet data in a slave packet contains the requested data.

Before the train is started up, the MVB must be statically configured on following items: (1) which data are requested by the master device in each period; (2) which slave devices receive the requested data. (3) usually a slave packet contains a set of data, the format of these data in a slave packet is also statically configured. Fig. 1 (a) shows an example of the slave packet.

III. Definitions

In this section, we present important definitions used in the paper.

Let X be the set of typed variables in a MVB system. For any $x \in X$, denote $T(x)$ the type of x. Three basic data types are supported in $VeRV$, i.e., $Integer, Boolean$, and $String$. For example, the type of v_{speed} in Fig. 1 is $Integer$.

Each variable in X represent a process data maintained by certain device in the system. Each time there comes a packet, the conveyed process data in the packet are extracted and assigned to the corresponding variables in X. Note that a packet may not convey all process data in the system. Given a packet ξ, denote $X_\xi \subseteq X$ the subset of variables whose data are conveyed in ξ. If a process data is not conveyed in the current packet, the corresponding variable keeps its original value.

Definition 1 *An MVB packet ξ is a full assignment to X, such that $\xi(x)$ is the data conveyed in the packet if $x \in X_\xi$, and $\xi(x) = x$ otherwise.*

At any time there is at most one packet being transmitted on the MVB. We call the packet currently being transmitted the *current packet*, and the packets already transmitted the *history packets*. In many cases, we need to know not only the data in the current packet, but also the data in the history packets.

Let X^h be the set of *history variables*. For any history variable $x^h \in X^h$, there is a variable $x \in X$, such that either (1) x^h is a copy of x and refers to the value of x in a history packet, or (2) x^h is a statistical variable based on the values of x in a number of history packets. Note that there may be more than one history variables in X^h corresponding to the same variable x in X.

Example 1 *Consider the packet example in Fig. 1, assume that we want to know the speed data in the latest packet, a history variable named $v^h_{latest\_speed}$ is then used to record this data. We cannot use the variable v_{speed} since it has already been used to record the speed in the current packet. If we want to know the number of all processed packets, we can use another history variable v^h_{count}.*

Definition 2 *An MVB* event *p is a Boolean expression on $X \cup X^h$. We call the MVB event p* happens *in the packet ξ if the value of p is true with the assignment of ξ, i.e., $p(\xi) = true$.*

Note that multiple Boolean expressions may be evaluated true by the assignment of a packet. In other words, more than one events may happen in a packet. Let *AP* be a finite set of all events that we are concerned about in the system. The concerned events *AP* may not involve all variables in X. Let X_{AP} be the set of all variables occurring in any event of AP, called the *support* of AP. Obviously $X_{AP} \subseteq X \cup X^h$.

Definition 3 *Let $\Sigma = 2^{AP}$ be the alphabet, a finite MVB* trace *is a finite sequence $a_1a_2\cdots a_n$, where $a_i \in \Sigma$ for $1 \le i \le n$. An infinite MVB* trace *is a infinite sequence $a_1a_2\cdots$, where $a_i \in \Sigma$ for $i \ge 1$. Denote the set of all finite MVB traces by Σ^*, and the set of all infinite MVB traces by Σ^ω.*

For any $a \in \Sigma$, a is a subset of *AP*, i.e. a subset of events. In other words, an MVB trace is a sequence of set of events. Let $\emptyset$ be the empty set of events, apparently $\emptyset \in \Sigma$.

Example 2 *Consider the packet in Fig. 1, assume we are interested in the set of following events:*

$$\begin{aligned} \mathit{NoSpeed} &:= (v_{speed} = 0) \\ \mathit{HighSpeed} &:= (v_{speed} \ge 120) \\ \mathit{NormalSpeed} &:= (v_{speed} > 0 \wedge v_{speed} < 120) \\ \mathit{HighTemp} &:= (v_{temp} > 27) \\ \mathit{LowTemp} &:= (v_{tempe} < 23) \\ \mathit{NormalTemp} &:= (v_{temp} \le 27 \wedge v_{temp} \ge 23) \\ \mathit{AcOn} &:= (b_{ac} = 1) \\ \mathit{AcOff} &:= (b_{ac} = 0) \end{aligned}$$

The support of above events is $\{v_{speed}, v_{temp}, b_{ac}\}$. Each time a packet comes, we extract the values of these three variables and evaluate above Boolean formulas. After processing the packet, we get a set of happened events. Assume the values in the current packet are:

$$v_{speed} = 100, v_{temp} = 28, b_{ac} = 1$$

Then the set of happened events is

$$\{\mathit{NormalSpeed}, \mathit{HighTemp}, \mathit{AcOn}\}$$

As the packet comes ever and again, we may get a MVB trace like: $\{\mathit{NormalSpeed}, \mathit{HighTemp}, \mathit{AcOn}\}$, $\{$HighSpeed, LowTemp, AcOff$\}$, $\{$HighSpeed, NormalTemp, AcOn$\}$, ...

Definition 4 *An MVB* property *is a function $\varphi : \Sigma^* \to \{validation, violation, unknown\}$ mapping each MVB trace to one of these three labels: validation, violation and unknown [11].*

A property can also be regarded as a classifier that partitions a set of MVB traces into three groups: the $validation$ traces, the $violation$ traces and the $unknown$ traces.

Example 3 *Consider a property: the train should never run at a speed over 120 km/s. This property can be described as: there is no HighSpeed event in any MVB trace. Given a finite MVB trace $\omega^* = a_1a_2\cdots a_n$, the trace is classified as follows: if there is an integer k ($1 \le k \le n$) such that HighSpeed $\in a_k$, ω^* is recognized as a violation trace; otherwise, it is recognized as an unknown trace. Note that for this (safety) property, ω^* will never be recognized as a validation trace. Because even the HighSpeed event has not happened in all encountered packets in ω^*, it is still uncertain whether this event will happen in a future packet.*

To formally specify the MVB properties, we employ the *past time linear temporal logic* (*ptLTL*) [9] [10]. We survey the syntax and semantics of *ptLTL* in the following.

Definition 5 (***Syntax of ptLTL***) *Let $p \in AP$, a ptLTL formula φ is inductively defined as follows:*

$$\varphi ::= true \mid p \mid !\,\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 U \varphi_2 \mid ()\,\varphi \mid (*)\varphi \mid \varphi_1 S \varphi_2$$

where $U, S, (), (*)$ stands for the "until", "since", "next" and "previous" temporal operators respectively.

For convenience, we also use the temporal operators of <>, [ ], <*>, [*] to represent "eventually", "always", "eventually in the past", "always in the past" respectively.

Definition 6 (***Semantics of ptLTL***) *Let $\omega = a_0a_1\cdots$ be an infinite MVB trace and i an integer, denote $\omega^i = a_ia_{i+1}\cdots$ the suffix of ω from the position i. The relation of $\omega, i \models \varphi$, i.e., the ptLTL formula φ holds for ω^i, is inductively defined as follows:*

$\omega, i \models true$		
$\omega, i \models !\, \varphi$	*iff*	$\omega, i \not\models \varphi$
$\omega, i \models p$	*iff*	$p \in a_i$
$\omega, i \models \varphi_1 \vee \varphi_2$	*iff*	$\omega, i \models \varphi_1$ *or* $\omega, i \models \varphi_2$
$\omega, i \models \varphi_1 U \varphi_2$	*iff*	$\exists k \geqslant i$ *with* $\omega, i \models \varphi_2$, *and* $\forall i \leqslant l < k$ *with* $\omega, l \models \varphi_1$
$\omega, i \models ()\varphi$	*iff*	$\omega, i+1 \models \varphi$
$\omega, i \models \varphi_1 S \varphi_2$	*iff*	$\exists 0 \leqslant j \leqslant i$ *with* $\omega, j \models \varphi_2$, *and* $\forall j < k \leqslant i$ *with* $\omega, k \models \varphi_1$
$\omega, i \models (*)\varphi$	*iff*	$\omega, i-1 \models \varphi$

Moreover, we say $\omega \models \varphi$ iff $\omega, 0 \models \varphi$. And we say two $ptLTL$ formulas φ_1 and φ_2 are *equivalent*, written $\varphi_1 \equiv \varphi_2$, if for any trace ω, $\omega \models \varphi_1$ holds if and only if $\omega \models \varphi_2$ holds.

Example 4 *The property in Example 3 can be rewritten in ptLTL as:*

$$[\,](not\ HighSpeed).$$

In the following, we discuss how does a finite MVB trace by classified by a $ptLTL$ formula [12].

Definition 7 *Given a finite MVB trace $\omega^* = a_1a_2\cdots a_n$ and an infinite MVB trace $\omega = b_1b_2\cdots$, we say ω is an* extension *of ω^* if $a_i = b_i$ holds for $1 \leq i \leq n$. Denote $E(\omega^*)$ the set of all extensions of ω^*.*

Definition 8 *Given a finite MVB trace ω^* and a ptLTL formula φ, ω^* is classified by φ*

1) *as a validation trace, if $\forall \omega \in E(\omega^*), \omega \models \varphi$;*
2) *as a violation trace, if $\forall \omega \in E(\omega^*), \omega \not\models \varphi$;*
3) *as an unknown trace, otherwise.*

Given a $ptLTL$ formula φ, we can generate an equivalent finite automaton, called the *monitor automaton*, such that a trace $\omega \models \varphi$ if and only if ω is accepted by this automaton [11]. Then we use this automaton to monitor the data transmitted on the MVB.

Definition 9 *A* monitor automaton *is a five-tuple $M = (S, s_0, \Sigma_M, \sigma, \gamma)$, where*

- *S is the set of states,*
- *$s_0 \in S$ is the initial state,*
- *Σ_M is the alphabet of M,*
- *$\sigma : S \times \Sigma_M \to S$ is the transition function, and*
- *$\gamma : S \to \{validation, violation, unknown\}$ is the output function.*

A monitor automaton can be represented as a state graph in a usual way. To distinguish the outputs of states, we use triple-circle nodes, double-circle nodes and single-circle nodes to depict the $validation$, $violation$ and $unknown$ states, respectively.

Example 5 *The monitor automaton for the property in Example 4 is shown in Fig. 4, where $S = \{s_0, s_1\}$, $\Sigma_M =$*

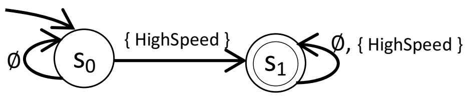


Fig. 4: An monitor automaton example

$\{\emptyset, \{HighSpeed\}\}$, σ is shown in the figure, and $\gamma(s_0) = unknown, \gamma(s_1) = violation$. Note that the only involved event in this property is HighSpeed. In other words, we have no need to identify all other events. If the event HighSpeed happens, we take the transition labelled with $\{HighSpeed\}$, otherwise we take the transition labelled with $\emptyset$.

IV. Work flow of Our Approach

Our $VeRV$ aims at runtime verification of the MVB. To use the $VeRV$, we first specify the packet format and the desired properties in our $VeSpec$ language, then a monitor will be automatically generated by the engine of $VeRV$ in a similar way as in [11]. The $VeSpec$ language will be introduced in the next section. This section focuses on the work flow of the generated monitor.

As shown in Fig. 5, the generated monitor works in following three steps:

- *Packet parsing*. In this step, the values of all relevant variables are extracted from the packet. The relevant variables are specified in the $VeSpec$ script and the packet parser can be generated automatically by our engine.
- *Events recognizing*. With the values of relevant variables, we check if there are any events happened in the current packet by evaluating the Boolean formula for each event. Here we do not need to check all events, instead we check only the events that are relevant to the desired property.
- *Property checking*. Let M be the monitor automaton of the desired property. Assume the automaton is currently on the state s. Given the set of happened events, we now decide which transition can the automaton take from s. If there is any transition which leads the automaton to a $violation$ state or a $validation$ state, we say the property is *violated* or *validated* by the trace. The monitor should report a *warning* massage or a *confirmation* message to the user. Otherwise, the trace is classified as $unknown$.

Above work flow repeats when here comes another packet.

Example 6 *Consider the monitor automaton in Fig. 4, the process of checking the corresponding property is shown in Fig. 6. The coming packet is shown in the left upper corner of the same figure. Here the only relevant variable for the property is v_{speed}. In the packet parsing step, we extract only the value of v_{speed}, which gives* 110. *Note also that the only relevant event for the property is HighSpeed. Thus in the events recognizing step, we need to evaluate the Boolean formula of HighSpeed only, whose value gives $false$. Therefore the set of happened events is empty. Assume the automaton (Fig. 4) is in the state of s_0 at present. With the empty set of happened events*

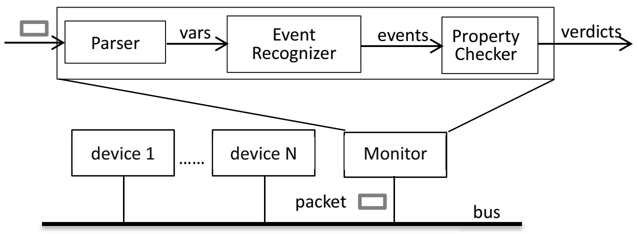


Fig. 5: The work flow of a monitor

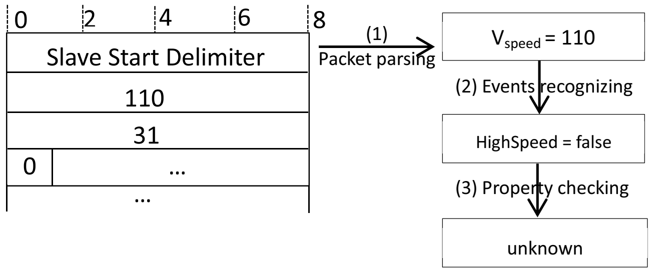


Fig. 6: The process of checking the example property

∅, the automaton can only take the self-transition. Note that s_0 *is an* $unknown$ *state, the current trace is thus recognized as an* $unknown$ *trace.*

V. *VeSpec* LANGUAGE

To achieve a general solution to the runtime verification of $ptLTL$ properties, we design a language, named $VeSpec$, for describing variables, events and temporal properties in the MVB system. With a script written in $VeSpec$, we are able to generate the corresponding monitor automatically using the $VeRV$ engine, in a similar way as in [11].

We design the language with the following characteristics: compactness, expressiveness, reusability and maintainability. Note that the testing engineers in train industry usually have little programming experience, the specification language thus should be easy to use. Moreover, the language should also be expressive enough to describe various properties. And the specifications should be reusable for different runtime analyzers of the same MVB system.

Fig. 7 shows the syntax of *VeSpec* language. The syntax is given in Backus-Naur Form (BNF), where each non-terminal is a constructor enclosed between the pair "<" and ">". A non-terminal is composed of one or several sub-constructors. Multiple choices in an expression are separated by the operator "|". A set of optical choices are surrounded by "[" and "]". And if an item appears zero or more times, it is enclosed by "{" and "}".

A. Specification

The top element of *VeSpec* is the *specification* entry. Each *VeSpec* script contains exactly one *specification* constructor. A *specification* constructor starts with its name – *spec-identifier*. Following the name, two parameters are used to specify the current packet: the *packet-name* and the *packet-length*. Note that the *packet-length* gives the number of bytes contained in this packet. These two parameters can be used in *code-segment* to define variables. For short, we use *data* and *length* to denote these two parameters.

```
<specification>        ::=  <spec-identifier>
                            "("  <packet-name> "," <packet-length> ")" "{"
                                <packet-types>
                                <vars>
                                <history-vars>
                                <events>
                                <properties>
                            "}"
<packet-types>         ::=  "PacketTypes" "=" <packet-type-name>
                            {"|" < packet-type-name>} ";"
<vars>                 ::=  "Vars" "{" { <vars-packet> } "}"
<vars-packet>          ::=  "Shared" "{" {<variable>} "}"
                            | <packet-type-name> "{" {<variable>} "}"
<variable>             ::=  <var-identifier> "{" <var-type> <var-extract-rule> "}"
<var-extract-rule>     ::=  <var-start-pos> <var-length>
                            | <var-update>
<var-type>             ::=  "Type:" <basic-data-type> ";"
<var-start-pos>        ::=  "StartPos:" <integer> ";"
<var-length>           ::=  "Length:" <integer> ";"
<var-update>           ::=  "Update" "{" <code-segment> "}"
<history-vars>         ::=  "HistoryVars" "{" {<history-variable>} "}"
<history-variable>     ::=  <var-identifier>
                            "{" <var-type> <var-initiation> <var-update> "}"
<var-initiation>       ::=  "Initiate:" <basic-type-value> ";"
<events>               ::=  "Events" "{" {event} "}"
<event>                ::=  <event-identifier> "=" <boolean-expression> ";"
<properties>           ::=  {<ptltl-property> { <handler> } }
<handler>              ::=  "@" <handler-type> "(" <focused-events> ")"
                            "{" <code-segment> "}"
<handler-type>         ::=  "violation" | "validation" | "unknown"
<focused-events>       ::=  <event-identifier> {"|" <event-identifier>}
                            | "*"
<basic-date-type>      ::=  "Integer" | "String" | "Boolean"
                            | "Integer[ ]" | "String[ ]" | "Boolean[ ]"
<basic-type-value>     ::=  <!-- Java Value of Basic data type  -->
<spec-identifier>      ::=  <!-- Java Identifier                -->
<packet-name>          ::=  <!-- Java Identifier                -->
<packet-length>        ::=  <!-- Java Integer Value             -->
<packet-type-name>     ::=  <!-- Java Identifier                -->
<var-identifier>       ::=  <!-- Java Identifier                -->
<integer>              ::=  <!-- Java Integer Value             -->
<code-segment>         ::=  <!-- Java Statements                -->
<event-identifier>     ::=  <!-- Java Identifier                -->
<boolean-expression>   ::=  <!-- Java  Boolean Expression       -->
<ptltl-property>       ::=  <!-- ptLTL formula                  -->
```

Fig. 7: Syntax of *VeSpec*

In *VeSpec*, a *specification* is composed of four parts: packet types, variables, events and properties. We discuss each of them in the following.

B. Packet Types

We use a *packet-types* constructor to express all types of packets transmitted in the system. Each *packet-types* contains one or several *packet-type-names*. A *packet-type-name* is a string identifying a packet type.

In different vehicle bus networks, the types of transmitted packets may be different. In a MVB system, there are two types of packets: the *Master* packet and the *Slave* packet. The packet types for the MVB are thus defined like this:

$$PacketType \;=\; Master \mid Slave.$$

C. Variables

Variables are described in the constructor of *vars*. Given two packets ξ_i and ξ_j which are of different types, $X_{\xi_i} = X_{\xi_j}$ may not hold. In other words, the sets of variables whose data are conveyed in ξ_i and ξ_j may not be the same. We define X_ξ for each type of packets in the *vars-packet* constructor. A *vars-packet* has an identifier *packet-type-name* and several *variable*s. If some variables are shared by packets of all types, the keyword *Shared* is used as their *packet-type-name*.

A variable is described in a *variable* constructor, which is constituted of a *var-identifier*, a *var-type* and a *var-extract-rule*. Three basic data types, i.e., $Integer$, $String$ and $Boolean$, and their array versions, i.e., $Integer[]$, $String[]$ and $Boolean[]$, are supported in $VeRV$. The constructor *var-extract-rule* is used to specify how to extract the value of this variable from a packet.

$VeRV$ provides two approaches to extract variables from the packets. One is called the *direct* method, which uses *var-start-pos* and *var-length* to indicate the start position and the occupied bits of the variable in the packet. Consider the packet as a bit vector, this value can be directed accessed from the packet. The other approach utilizes the *var-update* constructor which is constituted of a segment of Jave codes, to express the extraction rule. If both approaches are defined, $VeRV$ chooses the indirect approach by default.

Consider the example in Fig. 1, to specify the variables configured in this packet, we may use the following script:

```
Vars{
    Slave{
        speed{…}
        temperature{…}
        flag{…}
    }
    Master{…}
}
```

The value of v_{speed} is configured in the second byte of the packet. The following script illustrates the difference between these two extraction approaches:

```
speed{
    Type: Integer;
    StartPos: 8;
    Length: 8;
}
```

```
speed{
    Type: Integer;
    Update{
        speed = data[1] & 0xff;
    }
}
```

The direct approach is described in the left, while the indirect approach is shown in the right. In both cases, we are able to extract the value of v_{speed} from the second byte of the packet. Obviously the indirect method is more flexible and can express more complicated extraction rules. However, the direct approach is more compact when dealing with variables that occupy some continuous bits in the packets.

The *history-vars* constructor is used to specify the history variables. It consists of several *history-variable* constructors, each of which describes a history variable. A *history-variable* constructor is composed of an identifier *var-identifier* and three attributes: *var-type, var-initiation* and *var-update*. The initial value of a history variable is declared in *var-initiation*, which must conform to the *var-type*. *Var-update* is used to express the value extraction rule of a history variable.

For example, the history variable $v^h_{latest\_speed}$ in Fig. 6 can be declared as follows:

```
HistoryVars{
    latestSpeed{
        Type: Integer;   Initiate: 0;
        Update{ latestSpeed = speed; }
    } …
}
```

where *LatestSpeed* is an integer whose initial value is set to 0. Each time a packet comes, the *LatestSpeed* is updated with the value conveyed in the latest packet.

D. Events

The events are declared in the *events* constructor which is composed of a number of *event* constructors. As we mentioned before, an event is defined as a Boolean expression. The declaration of an *event* consists of two parts: an *event-identifier* and a *Boolean-expression*. The Boolean expression of an event is defined over variables and history variables.

E. Properties

The properties to be verified are declared in the *properties* constructor. The declaration of a property consists of a *ptLTL-property* and optional number of handlers. The *ptLTL-property* describes a formula in the $ptLTL$ logic. The *handler* constructor is composed of a start delimiter "@", a *handler-type* constructor, a *focused-events* constructor and a *code-segment* constructor. A *handler-type* takes a value in the following: *violation*, *validation* and *unknown*. A *focused-events* is a sequence of events separated by the operator "|". The code segment of the handler is executed, if the next state is with the same type of the *handler-type*, and one event in the *focused-events* happened.

For example, the first property discussed in Section I can be described as follows:

```
Properties{
    [ ]((HighTemp /\ AcOn) \/ (LowTemp /\ AcOff))
    @violation{
        System.out.println("temperature: "+ temperature);
        _RESET();
    }
}
```

The script declares a *violation* handler without the *focused-event*. Once the monitor automaton transits to a violation state, the handler is executed which prints the current temperature value and then reset the monitor to its initial state by the _RESET command.

```
MVBSpec(data, length){
    PacketTypes = Master | Slave;
    Vars{
        Shared{
            PacketType{    Type: String;
                Update{
                    PacketType = null;
                    if(data[0] & 0x20 == 0x00){ PacketType = "Master";
                    }else{ PacketType = "Slave"; }
                }
            }
        }
        Master{
            Fcode{ Type: Integer; StartPos: 8; Length: 4; }
            LogicAddr{ Type: Integer; StartPos: 12; Length: 12; }
        }
        Slave{ Ax1Speed{ Type: Integer; Update{…} } }
    }
    HistoryVars{
        _fcode{ Type: Integer; Initiate: 0;
            Update{ if(PacketType.equals("Master")) _fcode = Fcode; }
        }
    }
    Events{
        IsMasterPacket = {PacketType.equals("Master")}
        IsSlavePacket = {PacketType.equals("Slave")}
        IsProcessData = { Fcode >= 0 && Fcode <= 4 }
        RequireAx1Speed = { LogicAddr == 0x518 }
        IsSlavePacketRight = { length == (4 << _fcode) }
        IsAx1SpeedValid = { Ax1Speed >= 0 && Ax1Speed <= 1378}
    }
    Properties{
        [ ](IsSlavePacket => (*)IsMasterPacket )
        @violation{ print("Master packet missing!"); _RESET(); }
        [ ](IsMasterPacket /\ IsProcessData =>
          o (IsSlavePacket /\ IsSlavePacketRight ) )
        @violation{ print("Slave packet length error!"); _RESET(); }
        [ ](IsMasterPacket /\ IsProcessData /\ RequireAx1Speed =>
          o IsAx1SpeedValid)
        @violation{ print(_CURRENTEVENT() + "Ax1Speed:" + Ax1Speed);}
        @unknown(IsAx1SpeedValid){
            print(_CURRENTEVENT() + "Ax1Speed:" + Ax1Speed);}
    }
}
```

Fig. 8: The *VeSpec* script for the case

VI. Evaluation

To evaluate the *VeRV*, we applied it to a real tube train system. The results show the effectiveness of *VeRV* and the applicability of *VeSpec* to the MVB.

A. *VeSpec script*

A part of *VeRV* script for specifying the MVB in the real system is shown in Fig. 8. The name of specification is *MVBSpec*. We use *data* and *length* to denote the current packet and its length.

Like other MVB systems, two types of packets are transmitted in this system: the *Master* packets and the *Slave* packets. The *Master* packets convey values of two variables: *Fcode* and *LogicAddr*. If a master packet is used for requesting some process data, the logic address of the requested data is specified in the *LogicAddr*. Some *slave* packets convey the *Ax1Speed* value, which represents an axle's running speed. The value of *PacketType* is conveyed by all types of packets. It is thus declared as a *Shared* variable. The variables *Fcode* and *LogicAddr* are extracted from the packet directly. The values of *PacketType* and *Ax1Speed* are extracted in the indirect approach. Only one history variable is declared in the script: *_fcode*. It is used to record the value of *Fcode* in the latest master packet.

Five events are declared in the script. The *IsMasterPacket* event and the *IsSlavePacket* event specify if the current packet is a master packet or a slave packet, respectively. According to the protocol [1], if the value of *Fcode* in a master packet is between 0 and 4, this packet is used for sending a request for some process data, then the *IsProcessData* event is evaluated true. The *RequireAx1Speed* event is evaluated true if the master packets is used for requesting the process data *Ax1Speed*. Note that $0x518$ is the logic address of *Ax1Speed*. The *IsSlavePacketRight* event indicates if the length of the slave packet conforms to the Fcode value in the latest master packet. The *IsAx1SpeedValid* event specifies if the *Ax1Speed* value meets a predefined constraint.

Three properties are specified in the script:

1) *Master Packet Required.* In the MVB, a slave packet is used to give response to a master packet. Thus, if here comes a slave packet, the previous packet must be a master packet. If this property is violated, the monitor prints an error message and then resets the monitor automaton to its initial state.
2) *Slave Length Constraints.* Following a master packet, there is a slave packet. And the length of the slave packet must conform to the *Fcode* value in the previous master packet. If this property is violated, the monitor prints an error message, and the monitor automaton is reset.
3) *No Overspeed.* This property specifies a constraint on the speed of an axle. If the current master packet sends a request for the *Ax1Speed* value, the next packet must be a slave packet containing the *Ax1Speed* data and this data must be in the range of [0..1378]. If the property is violated, the handler prints the current events and the current *Ax1Speed* value. Or, if the property is unknown to be violated or validated, but the *IsAx1SpeedValid* event happened in the current packet, the same handler is executed.

B. *Experimental Results*

We obtained a log file of a long run of a real tube train system. We were told that a system failure occurs in this run, and we were asked to diagnose this failure. The log file contains 11,455,367 packets in total. We use the *VeRV* to generate a monitor based on the script in Fig. 8, and then use this monitor [2] to analyze this log file.

The experiments results are listed in Table I. We analyze these results in the following:

[2]Although the monitor is used offline in this experiment, it is essentially an online analyser.

TABLE I: Experimental results

Property	Counts of Violations
Master Packet Required	1
Slave Length Constraints	1,295,534
No Overspeed	0

1) *Master Packet Required.* From the result, one master packet is missing. After carefully inspection, we found that the log file starts with a record of a slave packet. This property is just violated on the first slave packet packet in the log file. Of course there is no packet before it, we thus consider it as a reasonable phenomenon.
2) *Slave Length Constraints.* As shown in Table I, there are totally 1,295,534 violations for this property. Two cases can lead to these violations: i) the slave packet is missing; ii) the length of the slave packet does not conform to the preceding master packet. We checked the variable *PacketType* in packets where a violation occurs, and found many cases that after a master packet, the next packet is not a slave packet. In other words, many slave packets are lost. We believe this is the main reason for these violations.
   Since the master packets were transmitted normally, we deduced that the packet receiving devices may go wrong. We thus analyzed the program run on this device and finally succeeded in finding a deadlock in the program. After fixing this bug, the same failure has never occurred.
3) *No Overspeed.* There is no violation for this property, which means the axle works normally.

Failure diagnosis in an industrial system is a complicated task, requiring comprehensive analysis and integrated application of various techniques. Many techniques, like signal processing, program analysis and data mining have been applied to failure diagnosis of this case. Our technique contributed in finding the abnormal behaviors of the packet receiving devices, which is the key for the final diagnosis. Thus the effectiveness of $VeRV$ is illustrated. The $VeRV$ is also very efficient. In checking all properties on the whole of the huge log file, it took less than 6.8 minutes.

VII. Related Works

In this section, we review four categories of research works related to our work.

Model Checking: Model checking [13] is a famous technique for correctness verification of systems and protocols. Some studies have been performed on the vehicle bus systems. In [14], the authors applied the Murφ verification system [15] to specify a controller area network protocol and succeeded in verifying 12 properties using model checking. Model checking is a kind of statical approaches, which cannot be applied at runtime. And model checking always faces the state explosion problem. Our work focuses on the property checking at runtime. And our technique scales up very well. Note here only the runtime trace (one trace) needs to be checked, the state space explosion problem is alleviated.

Runtime Verification: Runtime verification (RV) [16]–[18] is a light-weighted verification technique. The basic idea of RV is to verify properties with runtime information of systems, which is simple but effective. RV has been applied to various fields, such as medical systems [19]–[21], Java programs [11], [22], PCI bus [11] and *Communications Based Train Control* (CBTC) systems [20]. Typically researchers specify properties with finite state machine (FSM), linear temporal logic (LTL) or other formalisms, and then generate the runtime monitors automatically. We believe RV is a natural and satisfactory solution for verifying properties of the MVB systems. Note that the CBTC systems stuided in [20] are different from the vehicle bus systems. The CBTC system is used for communication at a higher level, among the track equivements, trains and train stations. It is not bus-based, and thus its techniques for extracting events, specifying properties, and generating monitors are different from MVB's. To the best of our knowledge, this is the first work on applying runtime verification to vehicle bus systems.

Protocol Analysis: Plenty of works have been done on protocol analysis, such as CSN.1 [23], PacketType [24], Binpac [25], Zube [4] and GAPA [3]. The former three tools are mainly designed for the packet parser generation. The user-defined handlers are supported in CSN.1 and GAPA, but with which only simple analysis on extracted fields can be defined. Zube [4] uses annotated ABNF to specify protocols. With a protocol specification, Zube is able to generate the protocol-handling layer for network applications. The last work, GAPA [3], is more relative to our works. GAPA proposes a language, called GAPAL, to describe the packet format, event handlers and protocol state machine. However, the main focus of GAPA is to specify and analyze the protocol state machine. In contrast, our focus is on the analysis of the transmitted data.

Network Anomaly Detection: The runtime monitoring technique has been widely applied in the rule-based network anomaly detection systems, which are used to detect intrusion through network data [26]. Bro [27] provides a language to specify security policies. When a network traffic stream comes, Bro reduces it into an events sequence and perform analysis like storing information, which is guided by the specified policies. STATL [28], SHIELD [29] and VeTo [30] apply state machine to specify attacks and vulnerable behavior. This state transition analysis is similar to protocol state monitoring.

Our work is different from above works on protocol analysis and network anomaly detection in following aspects:

1) *Goals.* Our goal is to analyze the temporal and data-concerned properties at runtime, while above works are mainly focused on the packet parsing, protocol analysis and anomaly detection, respectively.
2) *Language.* With the $ptLTL$, our $VeSpec$ language can express temporal properties easily. Other languages in above works are mainly designed for specifying rules or policies. The expressiveness of $VeSpec$ for specifying temporal properties is much more powerful.
3) *Target.* All above works targets the Internet, while our tool targets the vehicle bus networks.

VIII. Conclusion

In this paper, we proposed a runtime verification framework for the MVB. To the best of our knowledge, this is the first temporal and data-concerned analyzer for the MVB. We implemented *VeRV*, a prototype tool for our technique. A domain-specific language, *VeSpec*, is designed for specifying the packet formats and the temporal properties. With the *VeSpec* script, the *VeRV* engine generates the runtime analyzer automatically. The *VeSpec* language is compact, expressive, modular and reusable. We have applied *VeRV* to a real MVB system and succeeded in diagnosing a real failure in this system. The industry application illustrates the effectiveness and efficiency of our technique.

In the future, we plan to improve *VeRV* in two directions: (i) Only Java language is supported in the generated monitors, we plan to modify the engine of *VeRV* to generate monitors in more languages. (ii) Timing is an important property for the MVB. We are going to investigate the runtime timing analysis for the MVB systems.

Acknowledgment

This work was supported in part by the Chinese National 973 Plan (2010CB328003), the NSF of China (61272001, 60903030, 91218302), the Chinese National Key Technology R&D Program (SQ2012BAJY4052), the Importation and Development of High-Caliber Talents Project of Beijing Municipal Institutions (YETP0167), and the Tsinghua University Initiative Scientific Research Program.

References

[1] I. E. Commission *et al.*, "IEC 61375–1, part 1: Train communication network," 1999.

[2] H. Kirrmann and P. A. Zuber, "The IEC/IEEE train communication network," *IEEE Micro*, vol. 21, no. 2, pp. 81–92, 2001.

[3] N. Borisov, D. Brumley, H. J. Wang, J. Dunagan, P. Joshi, C. Guo, and I. Nanjing, "Generic application-level protocol analyzer and its language," in *NDSS*, 2007.

[4] L. Burgy, L. Réveillère, J. Lawall, and G. Muller, "Zebu: A language-based approach for network protocol message processing," *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 575–591, 2011.

[5] F. Systems, "PTS 402: Portable test system," http://www.farsystems.it.

[6] E.-E. Ltd., "Portable system tester," www.eke.com.

[7] Siemens, "TCN bus tester," http://www.siemens.com.

[8] Automation NetworkX GmbH, "TCNalyzer," http://tcnalyzer.net.

[9] F. Laroussinie, N. Markey, and P. Schnoebelen, "Temporal logic with forgettable past," in *Symposium on Logic in Computer Science*. IEEE Computer Society, 2002, pp. 383–383.

[10] O. Lichtenstein, A. Pnueli, and L. Zuck, *The glory of the past*. Springer, 1985.

[11] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu, "An overview of the MOP runtime verification framework," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 3, pp. 249–289, 2012.

[12] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for LTL and TLTL," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, p. 14, 2011.

[13] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT press, 1999.

[14] M. Van Osch and S. A. Smolka, "Finite-state analysis of the CAN bus protocol," in *Sixth IEEE International Symposium on High Assurance Systems Engineering*. IEEE, 2001, pp. 42–52.

[15] D. L. Dill, "The Murphi verification system," in *Proceedings of Computer Aided Verification*. Springer, 1996, pp. 390–393.

[16] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky, "Formally specified monitoring of temporal properties," in *Proceedings of the 11th Euromicro Conference on Real-Time Systems*. IEEE, 1999, pp. 114–122.

[17] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan, "Runtime assurance based on formal specifications," *Departmental Papers (CIS)*, p. 294, 1999.

[18] K. Havelund, "Using runtime analysis to guide model checking of java programs," in *Proceedings of SPIN Model Checking and Software Verification*. Springer, 2000, pp. 245–264.

[19] D. Arney, M. Pajic, J. M. Goldman, I. Lee, R. Mangharam, and O. Sokolsky, "Toward patient safety in closed-loop medical device systems," in *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*. ACM, 2010, pp. 139–148.

[20] L. Bu, Q. Wang, X. Chen, L. Wang, T. Zhang, J. Zhao, and X. Li, "Toward online hybrid systems model checking of cyber-physical systems' time-bounded short-run behavior," *ACM SIGBED Review*, vol. 8, no. 2, pp. 7–10, 2011.

[21] T. Li, F. Tan, Q. Wang, L. Bu, J.-N. Cao, and X. Liu, "From offline toward real time: A hybrid systems model checking and CPS codesign approach for medical device plug-and-play collaborations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 642–652, 2014.

[22] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan, "Java-MaC: a run-time assurance tool for java programs," *Electronic Notes in Theoretical Computer Science*, vol. 55, no. 2, pp. 218–235, 2001.

[23] M. Mouly, *CSN. 1 Specification, Version 2.0*. Cell & Sys, 1998.

[24] P. J. McCann and S. Chandra, "Packet types: abstract specification of network protocol messages," *ACM SIGCOMM Computer Communication Review*, vol. 30, no. 4, pp. 321–333, 2000.

[25] R. Pang, V. Paxson, R. Sommer, and L. Peterson, "binpac: A yacc for writing application protocol parsers," in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. ACM, 2006, pp. 289–300.

[26] S. Axelsson, "Intrusion detection systems: A survey and taxonomy," Technical report, Tech. Rep., 2000.

[27] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer networks*, vol. 31, no. 23, pp. 2435–2463, 1999.

[28] S. T. Eckmann, G. Vigna, and R. A. Kemmerer, "STATL: An attack language for state-based intrusion detection," *Journal of computer security*, vol. 10, no. 1, pp. 71–103, 2002.

[29] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier, "Shield: Vulnerability-driven network filters for preventing known vulnerability exploits," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 193–204, 2004.

[30] A. Lahmadi, O. Festor *et al.*, "Veto: An exploit prevention language from known vulnerabilities in sip services," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2010, pp. 216–223.