Abstract—Robot systems have been widely used in industry and also play an important role in human social life. Safety critical applications usually demand rigorously formal verification to ensure correctness. But for the increasing complexity of dynamic environments and applications, it is not easy to build a comprehensive model for the traditional offline verification. In this paper, we propose RobotRV, the first data-centered real-time verification approach for the robot system. Within this approach, a domain-specific language named RoboticSpec is designed to specify the complex application scenario of the robot system, the data packets transmitted in the robot system, and the safety critical temporal properties. Then, we develop an engine to automatically translate the RoboticSpec model into a real-time verifier. The generated verifier serves as an independent plug-in component for the runtime verification of concerned temporal properties. We applied the proposed approach to a real robot system. As presented in experiment results, our method detected potential failures, and improved the safety of robot system.

Index Terms—Collision avoidance, real-time data, robot system, runtime verification.

I. INTRODUCTION

ROBOT systems are playing an increasingly important role in our daily lives, and providing support in industry manufacturing [1], medical health care practices [2], and aviation control [3]. Those applications are life critical and have extremely high accuracy and safety requirements. Once safety requirements are violated, the consequences are serious or even fatal. It is highly desirable to validate those robot systems before deployment to ensure the safety as much as possible.

Traditionally, engineers apply testing techniques to the robot system for bug detection. Many testing softwares [4] and methods [5] are developed for the robot systems. But the simulation based testing techniques highly depend on the input patterns, and the coverage for some extra conditions. To solve the problems, traditional offline verification techniques, such as model checking [6] and theorem proving [7], widely used in verification of hardware and software, are brought into the validation of robot systems [8], [9]. Those offline techniques mainly focus on the verification of the control logic, and are not as concern as with runtime properties. Additionally, the increasing complexity of the dynamic environment and applications brings more challenges to current offline verification methods. More specifically, today’s robot systems involve dynamic running environments and complex communication among system components such as industry bus and wireless network. It is not easy to build a comprehensive and faithful system model for offline verification. Semiautomatic theorem proving is also hard for most engineers. Furthermore, building a comprehensive model may easily lead to state explosion because of the complex message sequences and advanced control logics.

In this paper, we propose RobotRV, a data-centered, real-time verification framework for the robot system, and aim to reduce the verification complexity through changing the verification from offline to runtime. First, we design a domain-specific language named RoboticSpec, to specify the application scenarios of the robot system, the packet format of the data transmitted within the robot system, and the data interacted with the dynamic environment, as well as the safety critical properties with respect to the application scenario. RoboticSpec is powerful in specifying real-time data related properties with the incorporation of past time linear temporal logic (pLTL) [10]. Then, we implement an engine to automatically translate the RoboticSpec model into a real-time verifier with the help of monitoring oriented programming (MOP) [11]. The generated real-time verifier can grab the desired data from the transmitted packets, verify those temporal properties, and provide warnings in case of violation. Additionally, the verifier serves as an independent and transparent lightweight plug-in without any change to the implementation of the original application, including the execution time.

Main contribution: The overall contributions of our work are summarised as follows.

1) To the best of our knowledge, RobotRV is the first lightweight real-time verification framework for robot

Rui Wang, Yingxia Wei, Houbing Song, Yu Jiang, Yong Guan, Xiaoyu Song, and Xiaojuan Li
system, which is independent of the underlying implementation and relies on the real-time data.

2) A domain specific language RoboticSpec is proposed to specify the application scenarios of robot systems, and corresponding tools are implemented for model specification and verifier generation.

3) The proposed framework has been applied to a real robot system and the results show the effects of the proposed method.

The rest of paper is organized as follows. First, we present related work in Section II and introduce some backgrounds of robot systems and ptLTL in Section III. Then, RobotRV design methodology is illustrated in Section IV, including the domain specification language and the design of the engine. Finally, the experiments’ evaluation on a real robot system is described in Section V and we conclude in Section VI.

II. RELATED WORK

In the past decades, robots have been applied in many kinds of fields. Recently there is lots of validation work for the robot systems.

Traditionally, most of them are related to testing and simulation. In [4], the authors apply multilevel testing strategies with the developed frameworks RoboFrame and MuRoSimF to identify potential errors. In [5], an embodied simulator combined with a reinforcement learning algorithm is applied to improve the real world odor location. In [12], the dynamic simulation of a cockroach-like hexapod robot is developed. Those techniques are efficient for debugging and exposing the basic function requirement inconsistencies depending on the quality of input test cases.

In order to overcome the incompleteness of testing, formal verification techniques [13], [14] are also applied to ensure the correctness of the robot system. Temporal logics are applied to specify motion planning tasks for mobile robots. The discrete plan that satisfies the temporal logic formula is synthesized by a model checking algorithm [8], [15]. Additionally, in [9], three obstacle-avoidance strategies are modeled by Markov Decision Process. Probabilistic model checking analyzes these strategies and finds the best solution in an uncertain dynamic environment. The model checking technique is automatic, but theorem proving can deal with more complex robot systems. Theorem Proving is used to verify a collision-free algorithm of dual-arm robot in [16]. Higher-Order Logic is applied to specify and verify robotic manipulation algorithms [17]. But this offline formal verification easily comes to the state space explosion problem. The model is always hard to comprehensively construct.

Recently, runtime verification [18], which is effective to verify real time temporal properties through obtaining real-time information from a running system, is used to detect the observed behaviors satisfying or violating certain properties in many fields, such as Medical systems [19], [20], and Vehicle Bus Systems [21].

Paper [22] uses a Domain-Specific Language (DSL) to declaratively specify a set of safety-related rules that the software must obey, as well as corresponding corrective actions that trigger when rules are violated. But the proposed DSL does not have formal semantics and cannot describe temporal properties. Some researchers have also tried to apply runtime verification in the security analysis of the robot operating system (ROS) [23] by providing a transparent monitoring infrastructure to monitor the commands and transmitted messages. This work is efficient with the ROS, and is dependent on the additional middleware inserted into the original communication entities and protocol. The generalization of this technique to other robot system architectures and applications is still under research. Our work focuses on the real-time data running on robot systems and safety properties related with these data.

III. BACKGROUND

In this section, we introduce some background of the robot systems and the property specification language ptLTL.

A. Communications in Robot Systems

Communications among the components of robot systems can be accomplished through a wireless or wired network. In this work, we take the wired CAN bus as an example [24], which efficiently supports real-time control with a very high level of security, and has been widely adopted in robots such as pipeline Crawl Robot [25], MarXbot [26], and the iCub humanoid robot [27].

We describe the standard message format of CAN bus as below. A message transmitted on the CAN bus is defined as a packet with a fixed format and limited length. There are two kinds of packet formats within the communication, standard packet, and extended packet, which differ in the length. The standard packet has 11 bits in the identifier field and the extended packet has 29 bits. For each kind of packet, it can be further divided into remote packet and data packet. When a node sends some information of another node, a remote packet would be sent to request a corresponding data packet which has the same identifier of the remote node.

The 13th bit of packet indicates the packet type. High level means data packet while low level means remote packet. For a successful communication, remote packet should go before the data packet and the length of data in data packet should be consistent with the remote packet. In actual applications, the packet may lose. If the data packet and the remote packet cannot match each other, errors may happen.

B. Syntax and Semantics of ptLTL

Runtime verification is a light-weight verification approach to assist traditional formal techniques. For verification, we use ptLTL to describe the properties with the past time modalities. We follow the syntax and semantics definition in [10].

ptLTL Syntax: Let $P = \{p_1, p_2, \ldots, p_i, \ldots, p_n\}$ be a set of atomic propositions, then ptLTL formulae are

$$\phi, \varphi ::= p_i | ¬\phi | (\phi \land \varphi) | \phi S \varphi | \phi U \varphi | T | F$$

where $¬, (\cdot), (\cdot), S, U$ stand for “negative,” “previous,” “next,” “since,” “until” temporal operators, respectively. $T$ means
always true while F means never true at all. What is more, there are other temporal operators which are useful for complex temporal properties with the past modalities. These operators can be defined by these basic operators. We define $<> \phi \equiv T U \phi$ which means $\phi$ will eventually be true, $[\phi] = \neg <\phi \neg \phi$ which means $\phi$ is always true in the future, $<\phi > \equiv T S \phi$ which means $\phi$ must either currently hold or have held somewhere in the previous trace, $[\phi] = \neg <\phi > <\phi >$ standing for always true in the past time. The classic LTL is a fragment of ptLTL. So, ptLTL can express all the specification that LTL describes.

ptLTL Semantics: Let $v$ be an infinite sequence $v = v_1 v_2 \ldots v_i \ldots$, with a mapping $\zeta : \forall i,v_i \rightarrow 2^\nu$ labeling atomic propositions that hold in each position $v_i$. With the structure path $(v, \zeta)$, a nonnegative integer $i$, and ptLTL formulae $\phi$ and $\varphi$, the relation that “$\phi$ holds at position $i$ in $v$ ” denoted as $v_i \models \phi$. It can be inductively defined as below:

\[
\begin{align*}
v, i \models p & \iff p \in \zeta(v_i) \\
v, i \not\models \phi & \iff v, i \not\models \phi \\
v, i \models \phi \land \varphi & \iff v, i \models \phi \land v, i \models \varphi \\
v, i \models (\phi) & \iff v, i + 1 \models \phi \\
v, i \models (\varphi) & \iff i > 0 \land v, i - 1 \models \phi \\
v, i \models \phi S \varphi & \iff v, m \models \varphi \text{ for some } 0 \leq m \leq i \\
 & \quad \text{s.t. } v, n \models \phi \text{ for all } m < n \leq i \\
v, i \models \phi U \varphi & \iff v, m \models \varphi \text{ for some } m \geq i \\
 & \quad \text{s.t. } v, n \models \phi \text{ for all } i \leq n < m.
\end{align*}
\]

Where two ptLTL formula $\phi$ and $\varphi$ are said to be equivalent, when condition “$v, i \models \phi$ iff $v, i \models \varphi$” is satisfied for all structure path $v$ and $i$. The equivalence relation between them can be denoted as $\phi \equiv \varphi$.

IV. VERIFICATION APPROACH

In this section, we describe how the work-flow of real-time data centered runtime verification cooperates with Robot systems. A domain specific language RoboticSpec is proposed to specify data, events, and properties of robot working scenarios. The semantics formalize the RoboticSpec models describing scenarios into the input sequence and automata for runtime verifier.

A. Verification Work-Flow

Runtime verification is a light-weight verification approach to assist formal techniques. The work-flow of the runtime verification framework of the robot system is depicted in Fig. 1. A domain specific language RoboticSpec is designed and specifies the packet formats. The data running in the robot system are formalized by RoboticSpec model. Temporal properties specify the safety requirements. Then we transform the RoboticSpec model to a runtime verifier with an engine based on MOP [11]. The generated automaton can monitor the real-time data. If a temporal property is violated, real-time warnings will be produced and shown on the monitor computer.

![Fig. 1. Runtime verification framework in the robot system.](image)

The runtime verifier presented in Fig. 2 is composed of three parts: data parser, event checker, and property checker. With these three parts, the runtime verifier can process the real-time data to get the runtime verification results as follows.

1) First, the Parser processes real-time data packet transmitted on the CAN bus of the robot system and gets the values. The packet formats, variables, and ways to extract values are defined by RoboticSpec. The parser is automatically generated by the developed engine based on the data part of RoboticSpec model.

2) Then, the Event Checker deals with the values of variables extracted in the first step. Based on the event definition of the boolean formula specified in the event part of RoboticSpec model, the events related to the desired properties are recognized.

3) Finally, in Property Checker, the events recognized in the previous step are applied to determine the transition of the automaton which is derived from the ptLTL formula specified in RoboticSpec. Once the upcoming event leads the automaton to a property-violation state, warnings are given to the robot system or alarm rings.

B. Domain Specific Language RoboticSpec

We propose a domain specific language RoboticSpec to model the scenarios of the robot system, including the real-time data, events, and temporal properties. Considering the limited experience of a robot system developer in runtime verification, the specific language should be easy to use and be able to sufficiently describe runtime data and a variety of temporal properties based on the ptLTL. While designing this language we considered the following characteristics: expressiveness and usability. The syntax of RoboticSpec is presented in BNF format. The constructor is enclosed in “<” and “>”. The operator "|" separates multiple choices in the expression. The constructors are surrounded by "{" and "}". Optional items are enclosed in square bracket.

**Scenario Specification:** Each RoboticSpec model stands for one scenario. The model starts with the reserved word Model followed by constructors model_id and model_body. model_id represents the name of a scenario model. In body of scenario
model model_body, packet_source, vars, history_vars, events, and properties are listed as constructors. The packet_source constructor is used to find the source to grab packets transmitted in the robot system. The vars constructor describes the current value of variables. The history_vars constructor defines the variables related to previous time. The events constructor specifies the significant event with a boolean value. The properties constructor formalizes the safety requirement.

\[
\text{Spec_model ::= 'Model'} \langle < \text{model_id} > \\
\langle < \text{model_body} > \rangle \\
\text{model_body ::= 'Source'} \langle < \text{packet_source} > \\
\langle < \text{vars} > \\
\langle < \text{history_vars} > \\
\langle < \text{events} > \\
\langle < \text{properties} > \\
\text{packet_source ::= < packet_name > < packet_length > \\
\text{model_id ::= 'a..z'A..Z'|z'|'(a..z)'A..Z' \\
|0'..9'|_) * \\
\text{packet_name ::= 'a..z'A..Z'|z'|'(a..z)'A..Z' \\
|0'..9'|_) * \\
\text{packet_length ::= integer.}
\]

Variables: Variables are specified in the vars constructor following a reserved words Vars. Variables include the identifier, type, and rule. If the variable both appears in data packet and remote packet, we attach keyword shared before its name of var_id. The var_id is the name of a variable. The type is similar to programming languages, mainly including integer, string, and boolean. The rule refers to the way to extract the value of a variable from a transmitted packet. In RoboticSpec, there are two ways to extract the value of variables from a packet. A direct way is to specify the start position and the bit length of the value in the packet. Another way is presented in the var_update constructor expressed as some Java codes.

\[
\text{vars ::= 'Vars': } \langle < \text{variable} > \rangle * \\
\text{variable ::= [shared] < var_type > \\
\text{var_type ::= 'integer'|'string'|'boolean' \\
'|integer']['string']['boolean']'} \\
\text{var_extract_rule ::= < var_pos > < var_length > \\
| < var_update > \\
\text{var_length ::= 'Position': integer \\
\text{var_update ::= 'Extract' \\
\{< - - Java Statements - - > \} \\
\text{var_id ::= 'a..z'A..Z'|z'|'(a..z)'A..Z' \\
|0'..9'|_') * .}
\]

Sometimes we need the variable value in the last packet. We cannot use the variable since it has been rewritten in the current packet. So we use the history_vars to define the previous value of variables. The constructor history_vars is defined with a reserved word HistoryVars and a set construct of his_variable. The his_var_update is similar to the rule of vars constructor to extract the value of history variable.

\[
\text{history_vars ::= 'HistoryVars': } \{ < \text{his_variable} > \} * \\
\text{his_variable ::= [shared] < his_var_type > \\
| < his_var_id > < his_var_update > \\
\text{his_var_type ::= 'integer'|'string'|'boolean' \\
'|integer']['string']['boolean']'} \\
\text{his_var_update ::= 'Update' < code >} \\
\text{his_var_id ::= 'a..z'A..Z'|z'|'(a..z)'A..Z' \\
|0'..9'|_') * \\
\text{code ::= < ! - Java Statements - > } .
\]

Events: The Events constructor specifies the boolean expression. Events is the reserved word for this constructor. The value of boolean expression denotes if the event has happened or not. T says the event happens. Otherwise, the event does not happen.

\[
\text{events ::= 'Events': } \{ < \text{event_id} > = '\text{'} \\
\langle < \text{event_id} > \rangle' * \\
\text{boolean_exp ::= < var_id > \\
| < his_vars_id > \\
| T | F \\
| < comput_exp > < \text{compare_op} > \\
\langle < \text{comput_exp} > \\
| < boolean_exp > ' & ' | ' |' \\
\langle < \text{boolean_exp} > \\
| '! | < boolean_exp > \\
\langle \{' | < boolean_exp > '}' \\
\text{comput_exp ::= < var_id > \\
| < his_vars_id > \\
\text{integer} \\
| < comput_exp > < \text{arith_op} > \\
\langle < \text{comput_exp} > \\
| \{' | < \text{comput_exp} > '}' \\
\text{compare_op ::= ' == ' | ' == ' | '< ' | '< ' | ' > ' | ' > ' \\
\text{arith_op ::= '/ | ' | '+' | '-' | '*' \\
\text{event_id ::= 'a..z'A..Z'|z'|'(a..z)'A..Z' \\
|0'..9'|_') * .}
\]

WANG et al.: FROM OFFLINE TOWARDS REAL-TIME VERIFICATION FOR ROBOT SYSTEMS 1715
Properties: The Properties constructor beginning with a reserved word Properties specifies the ptLTL formula and handler. The ptLTL formula is described based on the syntax and semantics of ptLTL in part B of Section III. The handler starts with a delimiter “@”. A handler type value is in the set \{validation, violation, unknown\}. When the corresponding event is triggered, the code of handler is executed. The related events are listed by focused_events.

\[
\text{properties ::= 'Properties : } \{ \langle \text{property_name} \rangle \text{'=} \langle \text{ptLTL_exp} >= \langle \text{handler} \rangle \} * \\
\text{ptLTL_exp ::= event_id} \\
| \neg \text{ptLTL_exp} \\
| \text{ptLTL_exp} \land \text{ptLTL_exp} \\
| \text{ptLTL_exp} \lor \text{ptLTL_exp} \\
| \text{ptLTL_exp} \Rightarrow \text{ptLTL_exp} \\
| \text{ptLTL_exp} \Rightarrow \text{ptLTL_exp} \\
| \langle \text{ptLTL_exp} \rangle \text{ptLTL_exp} \\
\text{handler ::= '@' } \langle \text{handler_type} \rangle \\
| '(' \langle \text{focused_events} \rangle ')\text{'}" - ! - Java Statements - "}" \\
\text{property ::= 'a' \langle \text{property_name} \rangle 'A' \langle \text{property_name} \rangle 'Z' \langle \text{property_name} \rangle 'a' \langle \text{property_name} \rangle 'A' \langle \text{property_name} \rangle 'Z' \langle \text{property_name} \rangle '0' \langle \text{property_name} \rangle '9' \langle \text{property_name} \rangle " \\
\text{handler_type ::= 'validation' } | \text{violation'} | \text{unknown'} \\
\text{focused_events ::= < event_id > } * .
\]

C. Formalization

The syntax of RoboticSpec has been defined above. We will give the semantics of RoboticSpec which indicates how to formalize data, event, and property. In other words, the semantics supply the mapping method from RoboticSpec model to a runtime verifier.

Data Formalization: The current data are the key to deciding the next step for the robot system. If the data exception is not detected, it may bring a fatal disaster. The data is formalized as a variable set V derived from vars constructor. The type of each data v derived from var_type constructor is denoted as T(v) \in \{integer, string, boolean, integer[], string[], boolean[]\}.

Similar to the current data, history data is formalized as a variable set V^h derived from the constructor history_vars. The type of the history data is the same as its current data. For each current data v \in V, there may be lots of history data derived in the past time nodes. In sampled packets, the data is formalized as below.

Formalization 1: \xi(v, p) is a full assignment on V and packets P, where \xi(v, p) is the value of data v contained in packet p, which is derived from the constructor var_extract_rule.

\[\xi^h \text{ is a full assignment to } V^h, \text{ where } \xi(v^h, p^h) \text{ is the value of data } v^h \text{ contained in packet } p^h, \text{ which is derived from } \text{the constructor } \text{his_var_update.} \]

where p is a new sampled packet, and p^h is a previously sampled packet before p.

Event formalization: Event formalization formalizes the boolean expressions in the constructor boolean_exp. The value of boolean expression denotes if the event has happened or not. Let E be a set of events derived from the constructor event, and the data set related to the event set is denoted as V_E, where V_E \subseteq V \cup V^h. The event is formalized as below.

Formalization 2: \forall e \in E, e \text{ is assigned by a boolean expression on } V_E. \text{ e is said to be happened when the assignment is evaluated to be true, which is denoted as } e(\xi(V_E, p)) = T.

Based on event formalization, event path should be defined as a sequence of sets, where each set \eta is the combination of events evaluated to be true and e_\eta is one of the event in \eta. It is a subset of all events contained in E. Then, the event path is formalized as follows:

Formalization 3: \eta^h is the group of all finite set sequence \eta (n = \eta_1, \eta_2, \ldots \eta_n), and \eta^\infty is the group of all infinite set sequence \eta (n = \eta_1, \eta_2, \ldots \eta_\infty). Each \eta_i contained in the set sequence is the event combination evaluated to be true, denoted as \eta_i = \{e_j | e_j(\xi(V_E, p_j)) = T\}, \text{ where } e_i \in E \text{ and } p_j \text{ is a packet in P}. \text{ Furthermore, if } \forall i \in [1,n], \eta_i = \eta_i', \text{ then } \eta^h \text{ is an extension of } \eta. \text{ All possible extensions of the finite path } \eta \text{ are denoted as } \Sigma(\eta).

In a direct current checking example, if the current is larger than 1120 mA, dangerous things may happen. Event High_DCurrent is defined to express this danger.

\[\text{High_DCurrent} = \text{DCurrent} > 1120.\]

Those events will be evaluated when there comes a data packet. A trace would be generated for continually sampled data packets.

Property Formalization: The property is described in the properties constructor. Property formalization is to map the property to an automaton which can judge if the property is satisfied or not. For original model checking, the evaluation is performed on infinite paths. In this context, runtime verification is different. It works on existing running systems and verifies the formula on the captured finite traces. As described in the paper [28], they introduce a three-valued semantics (validation, violation, and unknown) for each property and prove that these properties are monitorable. The corresponding handler types in the constructor handler_type are validation, violation, and unknown. The handler types are formalized as below.
Formalization 4: A handler derived from the pTTL formula \( \phi \) is a full assignment to \( \eta^* \) on the domain \{validation, violation, unknown\}, where \( \forall \eta \in \eta^* \), the assignment rule is as follows:

1. If \( \forall \eta \in \Sigma(\eta^*), \eta^* \models \phi \), then \( \phi(\eta) = \text{validation} \)
2. If \( \forall \eta \in \Sigma(\eta^*), \eta^* \not\models \phi \), then \( \phi(\eta) = \text{violation} \)
3. Else \( \phi(\eta) = \text{unknown} \).

The equivalent monitor automaton of the pTTL formula can compute the condition of the state transition. The translation from pTTL to automaton can be customized and automatically generated by MOP. The automaton formalized as below is used to monitor the event sequences defined on the real-time robot system, and the condition \( \eta^* \models \phi \) is satisfied when the corresponding path is accepted by the automaton. In general, after translating the pTTL property into a monitor automaton, the automata verification problem is a reachability analysis process, starting from the initial state, accepting the events, and decide the final state which is labeled with validation, violation, and unknown.

Formalization 5: A monitor automaton is defined as a tuple \( M = \langle S, s_0, \Sigma_M, \delta, O \rangle \), where

1. \( S = \{s_0, \ldots, s_n\} \) is the set of states,
2. \( s_0 \) is the initial state,
3. \( \Sigma_M \) is the alphabet of \( M \),
4. \( \delta = \{\delta_0, \ldots, \delta_n\} \) is the transition function,
5. \( O = \{o_0, \ldots, o_n\} \) is the output mapping the state to \{validation, violation, unknown\}.

The monitor automaton are generated by the following steps. First, we can get the nondeterministic Büchi automata \( A^\phi \) and \( A^{\neg \phi} \) accepting the infinite words which satisfy \( \phi \) and \( \neg \phi \) [29]. Then, using the evaluation rule, we obtain the corresponding nondeterministic finite automata \( \hat{A}^\phi \) and \( \hat{A}^{\neg \phi} \). Let \( \hat{A}^\phi \) and \( \hat{A}^{\neg \phi} \) be the deterministic version of \( \hat{A}^\phi \) and \( \hat{A}^{\neg \phi} \), which can be computed in a standard manner using the power-set construction. We construct the production automaton \( \bigcirc \hat{A} = \hat{A}^\phi \times \hat{A}^{\neg \phi} \). The monitor \( \hat{M} \) of \( \phi \) is the unique FSM obtained by minimizing the product automaton \( \hat{A} \). The correctness was proved in [28].

Here is an example. For the direct-current scenario there is a safety requirement that \( D\text{Current} \) value in the robot is not allowed to exceed its safe threshold. The event \( \text{High}_D\text{Current} \) describes the direct current exceeding the threshold. The requirement can be formalized as a pTTL formula \( \neg \text{High}_D\text{Current} \).

The generated monitor automaton for the above formula is described in Fig. 3. The automaton will start in the initial state \( S_0 \). If there is no event, the automaton will stay in the initial state \( S_0 \). When the event \( \text{High}_D\text{Current} \) labeled on the transition happens, the automaton will transit to violation state \( S_1 \). It will stay in \( S_1 \) if no event happens or \( \text{High}_D\text{Current} \) happens.

D. Implementation

After we model the application scenario of the robot system with RoboticSpec, an engine is needed to synthesize the RoboticSpec model into a real-time verifier. As described in subsection A, the engine should generate a data parser, event checker, and property checker. We develop an engine shown in Fig. 4 based on the semantics and formalization principle. The engine reads the variable part of RoboticSpec model written in the file and generate a data parser which can grab the desired variable from data packet on CAN bus. Event part of RoboticSpec model is used to generate the event checker. The event checker computes the value of events. The engine generates property checker from the property part of RoboticSpec model with the help of MOP. The property checker will verify the pTTL property and display warnings. The runtime verifier runs separately from the robot system. All the work is done in the accompanying monitor computer. The operation added to the original robot system is just reading data. Usually, this time will have very little effect on the real-time property of the original system.

V. CASE STUDY

A. Experiment Setup

We apply the proposed technique to a real OUR-1 robot for real-time verification. The OUR-1 is a lightweight industrial manufacturing robot and uses the CAN bus for communication. It is a six degrees of freedom manipulator. Fig. 5 illustrates a scenario that OUR-1 robot grabs objects from the table and puts them into different boxes. The triangles are put in one box and the rectangles are in the other one.

For this application scenario, we focus on the verification of properties which are safety critical for manufacturing applications. According to the discussion with the robot system
developer and domain experts, potential but very likely violations may happen in the following aspects:

1) The remote packet is not followed by a data packet. The connection will be established.
2) The ID value in the data packet is not equal to its value in the corresponding remote packet.
3) The direct current value of the data packet is out of the threshold of OUR-1 robot. It may lead to damage of the robot, products, or even the platform.

When the manipulator moves, each joint will rotate in a certain angle. Each joint has its limit of rotational degree of freedom. Even so, a collision between each joint could occur in the process of moving to the target position. For detecting the collision, we consider each joint an entity with a cylinder and a hemispherical end function.

In the algorithm, data are a series of information on every joint at a moment. They are angle value, position value on x-axis, y-axis, and z-axis, radius value, and angle with the coordinate plane. The algorithm is implemented in the controller. Some important function results are required to transfer on the CAN bus from position 24 to 27 in data packet. The function ANGLERIGHT is for checking the angle value. The functions CIRCLECYLINDER, CIRCLECYLINDER, and TWOCYLINDER checks collision if the projections of two arms are two cycles. If an arm’s projection is a cycle, the other is a polygon. CIRCLECYLINDER is used to check. If the projections are both polygon, we use TWO CYLINDER. In the RoboticSpec, some boolean variables are defined to record these function values. These variable names are used to define the events AngleRight, TwoCircle, CircleCylinder, and TwoCylinder. The values of these variables are used to evaluate these events to be true or false. If the collision is about to happen, the property will violate.

B. Specification

With the proposed language RoboticSpec, the above scenario can be described in the model as below.

Model: OUR_Robot
Source: Canbus 118
Vars:

bool RTR Position 12 Length 1;
integer RID Position 8 Length 4;
integer DID Position 8 Length 4;
integer DCurrent Position 16 Length 8;
integer Time Extract{initial 0;
    if (LastDCurrent>1120 & DCurrent>1120) Time=Time+1;
}
bool angleRight Position 24 Length 1;
bool twoCircle Position 25 Length 1;
bool circleCylinder Position 26 Length 1;
bool twoCylinder Position 27
Length 1;
}

HistoryVars:
    integer LastDCurrent Update{
        LastDCurrent = DCurrent;
    }

Events:
    RemotePacket = {RTR == F}
    DataPacket = {RTR == T}
    PacketMatch = {RID == DID}
    High_DCurrent = {DCurrent > 1120}
    Time_Out = {Time > 300}
    AngleRight = {angleRight == T}
    TwoCircle = {twoCircle == T}
    CircleCylinder = {circleCylinder == T}
    TwoCylinder = {twoCylinder == T}
    Stop = {DCurrent == 0}

Properties:
    p1 = [] (DataPacket => (*) RemotePacket)
        @violation {System.out.println('Remote packet is missing !');}
    p2 = [] (DataPacket / PacketMatch)
        @violation {System.out.println('data invalid');}
    p3 = [] (Time_Out => < *> High_DCurrent)
        @violation {System.out.println('DCurrent value exceed time constraint');}
    p4 = [] (Stop => < *> (TwoCircle Circle Cylinder TwoCylinder AngleRight))
        @violation {System.out.println('Collision may occur');}

OUR-Robot is the name of the scenario specification. Source includes the packet name and length. The boolean variable RTR stands for the type of packet. If the 12th bit in packet equals "0", it is a remote packet and RemotePacket event is true. Otherwise it is a data packet and DataPacket event is true. The integer variables RID and DID in the 8th position record the name of packet. The integer variable Time counts the number that LastDCurrent exceed 1120 and Dcurrent exceed 1120. Time can help estimate the event lasting time. Four boolean variables are defined later. They stand for four collision checking function results. A history variable is defined. The lastDCurrent records the DCurrent value of the last data packet.

In the events constructor, the PacketMatch event is true if RID in the remote packet equals the DID in the data packet. The High_DCurrent event is also true if the DCurrent value is beyond the defined constraint. The Time_Out event means the time variable is larger than a certain value. The event Stop represents the robot stopped moving. If the DCurrent equals 0, we know the robot stops.

In the property constructor, four requirements are formulated. Property P1 describes that the remote packet always arrived before data packet. Property P2 means the ID of data packet should always match with the remote packet. Property P3 says the direct current value cannot exceed the range for more than 3 s. This is hard to express in real time in pLTL logic. But we can turn this problem to an approximate one. We know the cycle time and the number that High_DCurrent happened continuously. We can compute the total by multiplying. Property P4 means if the two arms collide each other, the arm should stop moving. The arm collision depends on the four events. If one of the events happened, the collision will happen.

C. Verification Result

Based on RobotRV, we can get the verifier for the properties and plug it into the robot system for runtime monitoring. We test the generated monitor by continuously obtaining two thousand packets of OUR-1. The experiment results are shown in Table I. The Property column lists the properties used to verify in our experiments. The Packet Number is the count of packets, including remote packet and data packet. The last Violation Number column is the number of violations on the related properties.

From the experimental results, we can see that occasional violations happen, which is not easy for detection by human semiautomatic monitoring. The first property p1 is just violated when the first two packets are the data packet. In our code, we label the type of every packet. Based on trace-back analysis of the stored packet, we find that the first two packets are proved to be data packet. The remote packet is missing.

Property p2 checks the data consistency in remote packet and data packet. We find one violation. There are two reasons for this violation. Packet missing is one reason. The other reason is the disturbance from the environment. We checked the packet where violation happened, and found that RID in remote packet is not equal to DID in data packet. One bit in RID is changed because of unpredictable reason.

The third property p3 violates one time. It shows that the Direct Current value is out of the range for more than 3 s. At the same time, warning is given to facilitate the operator. While doing this experiment, we stopped the robot from moving by holding the arms. When the movement stops, the current will rise and exceed the threshold for a few seconds. Our runtime verifier discovered this unusual condition and give warning successfully. This is important while the robot is working with a human. It can avoid the potential danger.
The fourth property $p4$ violates zero times. It shows that no double arms collisions happen in the process. That’s the fact.

The failure diagnosis for robot system is a complex work. Our method can monitor the packet sending and receiving on CAN bus and find abnormal behaviors. According to the results, it is reasonable to draw the conclusion that our framework helps to ensure the safety of the robot system.

VI. CONCLUSION

In this paper, we proposed a verification framework RobotRV for the robot system to implement the verification from offline to real-time. The domain specific language RoboticSpec was designed to specify the application scenario of the robot system. We designed an engine which can automatically generate runtime verifier from the model specified by RoboticSpec. The proposed technique was applied to a real robot system and successfully diagnosed the property violation. Domain specifications are powerful enough to specify these complex properties.

In the future, we will continue our work in the following ways. Having verified the real-time data transmitted on the CAN bus of the robot system, we will generalize it to other communication bus based robot systems. More kinds of logics are planned to support the runtime verification framework.

REFERENCES


Rui Wang received the B.S. degree in computer science from Xi’an Jiaotong University, Xi’an, China, in 2004. She received the Ph.D. degrees in computer science from Tsinghua University, Beijing, China, in 2012. She is currently an Associate Professor in the College of Information Engineering, Capital Normal University, Beijing, China. Her current research interests include formal verification and their applications in embedded systems.

Yingxia Wei received the M.S. degree in computer technology from the College of Information Engineering of Capital Normal University, Beijing, China, in 2017. Her research interests include security of CPS and formal verification.
Houbing Song (M’12–SM’14) received the M.S. degree in civil engineering from the University of Texas, El Paso, TX, USA, in 2006 and the Ph.D. degree in electrical engineering from the University of Virginia, Charlottesville, VA, USA, in 2012.

He is currently an Assistant Professor with the Department of Electrical, Computer, Software, and Systems Engineering, Embry-Riddle Aeronautical University, Daytona Beach, FL, USA, and the Director of the SONG Lab. His current research interests include cyber-physical systems, intelligent transportation systems, wireless communications and networking, and optical communications and networking.

Yu Jiang received the B.S. degree in software engineering from Beijing University of Posts and Telecommunication, Beijing, China, in 2010, and the Ph.D. degree in computer science from Tsinghua University, Beijing, China, in 2015.

He is currently an Assistant Professor with Tsinghua University. His current research interests include domain specific modeling, formal computation model, formal verification and their applications in embedded systems.

Yong Guan received the Ph.D. degree in communication and information systems from the College of Mechanical Electronic and Information Engineering, China University of Mining and Technology, Xuzhou, China, in 2004.

He is currently a Professor with Capital University, Bexley, OH, USA. His research interests include formal verification of embedded system design. He is a member of the Chinese Institute of Electronics Embedded Expert Committee.

Xiaoyu Song received the Ph.D. degree in computer science from the University of Pisa, Pisa, Italy, in 1991.

From 1992 to 1998, he was on the faculty of the University of Montreal, Montreal, QC, Canada. He joined the Department of Electrical and Computer Engineering, Portland State University, Portland, OR, USA, in 1998, where he is currently a Professor. He was an editor of IEEE TRANSACTIONS ON VLSI SYSTEMS and IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS. He was awarded an Intel Faculty Fellowship from 2000 to 2005. His research interests include formal methods, design automation, and embedded systems.

Xiaojuan Li received the Ph.D. degree in mechanical and electronic engineering from China Agricultural University, Beijing, China, in 1999.

She is currently a Professor in the College of Information Engineering, Capital Normal University, Beijing, China. Her current research interests include formal verification and their applications in embedded systems.