From Offline Towards Real-Time Verification for Robot Systems

Rui Wang, Yingxia Wei, Houbing Song, Yu Jiang, Yong Guan, Xiaoyu Song, and Xiaojuan Li

***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.

Manuscript received May 31, 2017; revised September 8, 2017, November 21, 2017, and December 20, 2017; accepted December 22, 2017. Date of publication January 3, 2018; date of current version April 3, 2018. This work was supported in part by the National Natural Science Foundation of China (61303014, 61572331, 61472468, 61373034), in part by the National Key Technology Research and Development Program (2015BAF13B01), in part by National Key R & D Plan (2017YFC0806700, 2017YFB1301100), and in part by the the Project of Beijing Municipal Science and Technology Commission (LJ201607). Paper no. TII-17-1176. *(Corresponding authors: Yong Guan and Yu Jiang.)*

R. Wang, Y. Wei, Y. Guan, and X. Li are with the Beijing Advanced Innovation Center for Imaging Technology, Capital Normal University, Beijing 100048, China (e-mail: rwang04@cnu.edu.cn; 13521563226@163.com; guanyong@cnu.edu.cn; lixj66@gmail.com).

H. Song is with the Department of Electrical, Computer, Software, and Systems Engineering, Embry-Riddle Aeronautical University, Daytona Beach, FL 32114 USA (e-mail: h.song@ieee.org).

Y. Jing is with the School of Software, Tsinghua University, Beijing 100084, China (e-mail: jiangyu198964@126.com).

X. Song is with the Portland State University, Portland, OR 97201 USA (e-mail: songx@pdx.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TII.2017.2788901

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 problem, 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. Semiautomatical 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 realtime data related properties with the incorporation of past time linear temporal logic (ptLTL) [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

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 middle-ware 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 needs 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

$$\phi, \varphi ::= p_i | \neg\phi | (*)\phi | ()\phi | \phi \wedge \varphi | \phi S \varphi | \phi U \varphi | T | F$$

where $\neg$, $(*)$, $()$, 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 $<> \phi = \mathrm{T}\, U\phi$ which means ϕ will eventually be true, $[]\phi = \neg <> \neg\phi$ which means ϕ is always true in the future, $<*> \phi = \mathrm{T}\, S\phi$ which means ϕ must either currently hold or have held somewhere in the previous trace, $[*]\phi = \neg <*> \neg\phi$ 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. Some safety properties are more easily expressed in terms of past than the future. The ptLTL is expressive, modular, and easy-to-use. One can express properties in a modular way.

ptLTL Semantics: Let υ be an infinite sequence $\upsilon = \upsilon_1\upsilon_2\ldots\upsilon_i\ldots$, with a mapping $\zeta : \forall i, \upsilon_i \longrightarrow 2^P$ labeling atomic propositions that hold in each position υ_i. With the structure path (υ, ζ), a nonnegative integer i, and ptLTL formulae ϕ and φ, the relation that "ϕ holds at position i in υ" denoted as $\upsilon, i \models \phi$. It can be inductively defined as below:

$$\begin{array}{ll}
\upsilon, i \models p & \text{iff } p \in \zeta(\upsilon_i)\\
\upsilon, i \models \neg\phi & \text{iff } \upsilon, i \not\models \phi\\
\upsilon, i \models \phi \wedge \varphi & \text{iff } \upsilon, i \models \phi \text{ and } \upsilon, i \models \varphi\\
\upsilon, i \models ()\phi & \text{iff } \upsilon, i+1 \models \phi\\
\upsilon, i \models (*)\phi & \text{iff } i > 0 \text{ and } \upsilon, i-1 \models \phi\\
\upsilon, i \models \phi S \varphi & \text{iff } \upsilon, m \models \varphi \text{ for some } 0 \le m \le i\\
& \quad \text{s.t. } \upsilon, n \models \phi \text{ for all } m < n \le i\\
\upsilon, i \models \phi U \varphi & \text{iff } \upsilon, m \models \varphi \text{ for some } m \ge i\\
& \quad \text{s.t. } \upsilon, n \models \phi \text{ for all } i \le n < m.
\end{array}$$

Where two ptLTL formula ϕ and φ are said to be equivalent, when condition "$\upsilon, i \models \phi$ iff $\upsilon, i \models \varphi$" is satisfied for all structure path υ 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 specification 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.

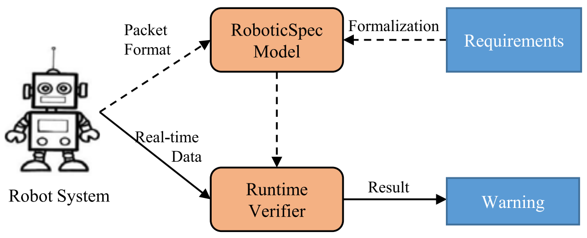

Fig. 1. Runtime verification framework in the robot system.

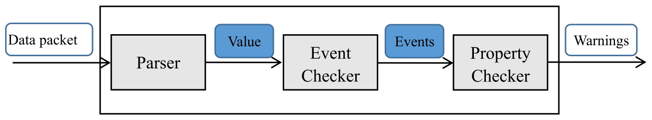

Fig. 2. Work-flow of runtime verifier.

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.

$$\begin{aligned}
Spec\_model &::= \ 'Model:' < model\_id > \\
&\quad \{< model\_body >\} \\
model\_body &::= \ 'Source:' < packet\_source > \\
&\quad < vars > \\
&\quad < history\_vars > \\
&\quad < events > \\
&\quad < properties > \\
packet\_source &::= \ < packet\_name > < packet\_length > \\
model\_id &::= \ ('a'..'z' | 'A'..'Z' | '\_')('a'..'z' | 'A'..'Z' \\
&\quad | '0'..'9' | '\_') * \\
packet\_name &::= \ ('a'..'z' | 'A'..'Z' | '\_')('a'..'z' | 'A'..'Z' \\
&\quad | '0'..'9' | '\_') * \\
packet\_length &::= \ integer.
\end{aligned}$$

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.

$$\begin{aligned}
vars &::= \ 'Vars:' \ \{< variable >\} * \\
variable &::= \ [shared] < var\_type > \\
&\quad < var\_id > < var\_extract\_rule > \\
var\_type &::= \ 'integer' | 'string' | 'boolean' \\
&\quad | 'integer[]' | 'string[]' | 'boolean[]' \\
var\_extract\_rule &::= \ < var\_pos > < var\_length > \\
&\quad | < var\_update > \\
var\_pos &::= \ 'Position:' \ integer \\
var\_length &::= \ 'Length:' \ integer \\
var\_update &::= \ 'Extract' \\
&\quad \{<! - -Java\,Statements - - >\} \\
var\_id &::= \ ('a'..'z' | 'A'..'Z' | '\_')('a'..'z' | 'A'..'Z' \\
&\quad | '0'..'9' | '\_') * .
\end{aligned}$$

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.

$$\begin{aligned}
history\_vars &::= \ 'HistoryVars:' \ \{< his\_variable >\} * \\
his\_variable &::= \ [shared] < his\_var\_type > \\
&\quad < his\_var\_id > < his\_var\_update > \\
his\_var\_type &::= \ 'integer' | 'string' | 'boolean' \\
&\quad | 'integer[]' | 'string[]' | 'boolean[]' \\
his\_var\_update &::= \ 'Update' \{< code >\} \\
his\_var\_id &::= \ ('a'..'z' | 'A'..'Z' | '\_')('a'..'z' | 'A'..'Z' \\
&\quad | '0'..'9' | '\_') * \\
code &::= \ <! - Java\,Statements - > .
\end{aligned}$$

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.

$$\begin{aligned}
events &::= \ 'Events:' \ \{< event\_id > '=' \\
&\quad < boolean\_exp >\} * \\
boolean\_exp &::= \ < var\_id > \\
&\quad | < his\_vars\_id > \\
&\quad | T \, | \, F \\
&\quad | < comput\_exp > < compare\_op > \\
&\quad < comput\_exp > \\
&\quad | < boolean\_exp > ' \&' \ | \ ' || ' \\
&\quad < boolean\_exp > \\
&\quad | '!' < boolean\_exp > \\
&\quad | '\{' < boolean\_exp > '\}' \\
comput\_exp &::= \ < var\_id > \\
&\quad | < his\_vars\_id > \\
&\quad | integer \\
&\quad | < comput\_exp > < arithe\_op > \\
&\quad < comput\_exp > \\
&\quad | '\{' < comput\_exp > '\}' \\
compare\_op &::= \ ' ==' \ | '! =' \ | ' <' \ | ' \geq' \ | ' \leq' \ | ' >' \\
arithe\_op &::= \ '/' | '\%' | ' +' \ | ' -' \ | ' *' \\
event\_id &::= \ ('a'..'z' | 'A'..'Z' | '\_')('a'..'z' | 'A'..'Z' \\
&\quad | '0'..'9' | '\_') * .
\end{aligned}$$

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*.

$$\begin{aligned}
properties &::= \,'Properties :' \, \{< property\_name >'=' \\
&\quad < ptLTL\_exp >< handler >\} * \\
ptLTL\_exp &::= event\_id \\
&\quad |\neg ptLTL\_exp \\
&\quad |\, ptLTL\_exp \wedge ptLTL\_exp \\
&\quad |\, ptLTL\_exp\, S\, ptLTL\_exp \\
&\quad |\, ptLTL\_exp\, U\, ptLTL\_exp \\
&\quad |\, [*]\, ptLTL\_exp \,|\, [\,]\, ptLTL\_exp \\
&\quad |\, <*>\, ptLTL\_exp \,|\, <>\, ptLTL\_exp \\
&\quad |\, (*)\, ptLTL\_exp \,|\, (\,)\, ptLTL\_exp \\
handler &::= \,'@' < handler\_type > \\
&\quad '(' < focused\_events > ')' \\
&\quad '' <! - Java\, Statements - >'' \\
property\_name &::= ('a'..'z'|'A'..'Z'|'\_')('a'..'z'|'A'..'Z'| \\
&\quad 0'..'9'|'\_') * \\
handler\_type &::= \,'validation'|'violation'|'unknown' \\
focused\_events &::= \,< event\_id > *.
\end{aligned}$$

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 $\mathbb{V}$ 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 $\mathbb{V}^h$ derived from the constructor *history_vars*. The type of the history data is the same as its current data. For each current data $v \in \mathbb{V}$, 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 $\mathbb{V}$ and packets $\mathbb{P}$, where $\xi(v, p)$ is the value of data v contained in packet p, which is derived from the constructor var_extract_rule. ξ^h is a full assignment to $\mathbb{V}^h$, where $\xi^h(v^h, p^h)$ is the value of data v^h contained in packet p^h, which is derived from the constructor 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 $\mathbb{E}$ be a set of events derived from the constructor *event*, and the data set related to the event set is denoted as $\mathbb{V}_\mathbb{E}$, where $\mathbb{V}_\mathbb{E} \subseteq \mathbb{V} \cup \mathbb{V}^h$. The event is formalized as below.

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

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

Formalization 3: η^* is the group of all finite set sequence $\eta\,(\eta = \eta_1\eta_2\eta_3 \ldots \eta_n)$, and η^τ is the group of all infinite set sequence $\eta'\,(\eta' = \eta_1'\eta_2'\eta_3' \cdots)$. Each η_i contained in the set sequence is the event combination evaluated to be true, denoted as $\eta_i = \{e_j | e_j(\xi(\mathbb{V}_\mathbb{E}, p_i)) = T\}$, where $\eta_i \in 2^\mathbb{E}$ and p_i is a packet in $\mathbb{P}$. Furthermore, if $\forall i \in [1, n], \eta_i = \eta_i'$, then η' is an extension of η. All possible extensions of the finite path η 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.

$$High\_DCurrent = \; DCurrent > 1120.$$

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.

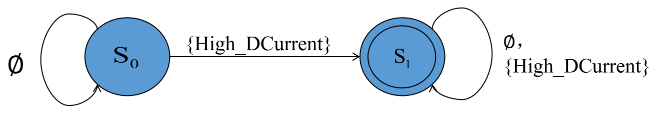

Fig. 3. Monitor automaton of direct current.

Formalization 4: A handler derived from the ptLTL formula ϕ is a full assignment to η^* 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* $\phi(\eta) = validation$
2) *If* $\forall \eta' \in \Sigma(\eta), \eta' \nvDash \phi$, *then* $\phi(\eta) = violation$
3) *Else* $\phi(\eta) = unknown$.

The equivalent monitor automaton of the *ptLTL* formula can compute the condition of the state transition. The translation from *ptLTL* 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 ptLTL 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 $M = \langle S, s_0, \sum_M, \delta, O \rangle$, where

1) $S = \{s_0, \ldots, s_n\}$ *is the set of states,*
2) s_0 *is the initial state,*
3) $\sum_M$ *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, and unknown*}.

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

Here is an example. For the direct-current scenario there is a safety requirement that *DCurrent* value in the robot is not allowed to exceed its safe threshold. The event *High_DCurrent* describes the direct current exceeding the threshold. The requirement can be formalized as a *ptLTL* formula $[\,](\neg High\_DCurrent)$.

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 *High_DCurrent* 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 *High_DCurrent* 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 ptLTL 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.

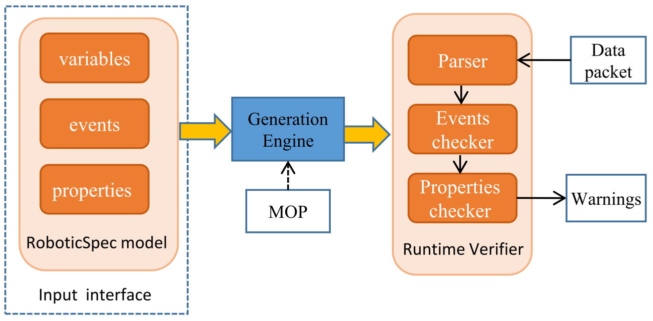

Fig. 4. Structure of parsers implemented by Java code in RobotRV.

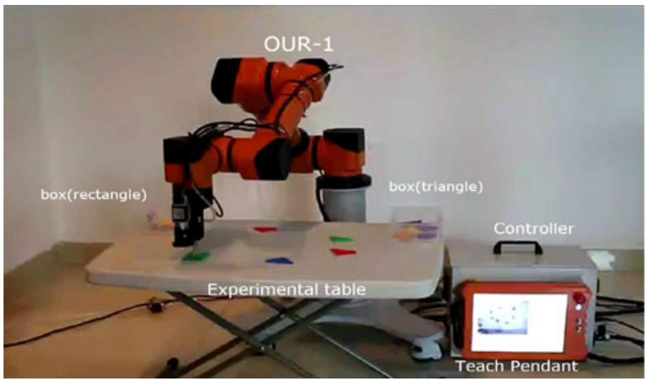

Fig. 5. Real scenario of OUR-1 robot.

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 both ends with hemisphere. By the method of orthogonal projection, the algorithm determines whether interference happened between joints. The algorithm shows as follows.

```
1: function COLLISION-DETECTION($(data)$)
2: $ANGLERIGHT\ (data)$
3: $TWOCIRCLE\ (data)$
4: $CIRCLECYLINDER\ (data)$
5: $TWOCYLINDER\ (data)$
6: end function
7:
8: function ANGLERIGHT$(data)$
9:     $result \leftarrow false$
10:    for $i = 0 \rightarrow 5$ do
11:        if $data >= -\pi\ and\ data <= \pi$ then
12:            $result \leftarrow true$
13:        end if
14:    end for
15: end function
16:
17: function TWOCIRCLE$(data)$
18:    $result \leftarrow false$
19:    for $i = 0 \rightarrow 5$ do
20: $code\ whether\ orthogonal\ projection\ of\ two\ circles$
    $and\ one\ polygon\ is\ interference$
21:        $result \leftarrow true$
22:    end for
23: end function
24:
25: function CIRCLECYLINDER$(data)$
26:    $result \leftarrow false$
27:    for $i = 0 \rightarrow 5$ do
28: $code\ whether\ orthogonal\ projection\ of\ one\ circle$
29: $and\ one\ pologon\ is\ interference$
30:        $result \leftarrow true$
31:    end for
32: end function
33:
34: function TWOCYLINDER$(data)$
35:    $result \leftarrow false$
36:    for $i = 0 \rightarrow 5$ do
37: $code\ for\ Homogeneous\ transformation$
38:            $data \leftarrow data1$
39: $twoCircle\ (data)1\ circleCylinder\ (data)1$
40:            $result \leftarrow true$
41:                end for
42:    if $result! = true$ then
43: $code\ for\ Homogeneous\ transformation$
44:            $data \leftarrow data2$
45: $twoCircle\ (data)2\ circleCylinder(data)2$
46:            $result \leftarrow true$
47:    end if
48: 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 *TWOCIRCLE* 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 *TWOCYLINDER*. 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(DCurrent){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.

TABLE I
EXPERIMENTAL RESULTS

Property	Packet Number	Violation Number
p1	2000	2
p2	2000	1
p3	2000	1
p4	2000	0

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 ptLTL 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

[1] D. Gu and H. Hu, "Neural predictive control for a car-like mobile robot," *Robot. Auton. Syst.*, vol. 39, no. 2, pp. 73–86, 2002.

[2] R. H. Taylor and D. Stoianovici, "Medical robotics in computer-integrated surgery," *IEEE Trans. Robot. Autom.*, vol. 19, no. 5, pp. 765–781, Oct. 2003.

[3] M. Summers, "Robot capability test and development of industrial robot positioning system for the aerospace industry," *SAE Trans.*, vol. 114, no. 1, pp. 1108–1118, 2005.

[4] S. Petters, D. Thomas, M. Friedmann, and O. Von Stryk, "Multilevel testing of control software for teams of autonomous mobile robots," In *Simulation, Modeling, and Programming for Autonomous Robots*. Berlin, Germany: Springer, 2008, pp. 183–194.

[5] A. T. Hayes, A. Martinoli, and R. M. Goodman, "Swarm robotic odor localization: Off-line optimization and validation with real robots," *Robotica*, vol. 21, no. 4, pp. 427–441, 2003.

[6] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. Cambridge, MA, USA: MIT press, 1999.

[7] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artif. Intell.*, vol. 2, no. 3, pp. 189–208, 1972.

[8] G. E. Fainekos, H. Kress-Gazit, and G. J. Pappas, "Temporal logic motion planning for mobile robots," in *Proc. IEEE Robot. Autom., Int. Conf.*, 2005, pp. 2020–2025.

[9] R. Wang, M. Wang, Y. Guan, and X. Li, "Modeling and analysis of the obstacle-avoidance strategies for a mobile robot in a dynamic environment," *Math. Problems Eng.*, vol. 2015, 2015, Art. no. 837259.

[10] François Laroussinie, Nicolas Markey, and Philippe Schnoebelen, "Temporal logic with forgettable past," in *Proc. Logic Comput. Sci., 17th Annu. Symp.*, 2002, pp. 383–392.

[11] P. O'Neil Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu, "An overview of the mop runtime verification framework," *Int. J. Softw. Tools Technol. Transfer*, vol. 14, no. 3, pp. 249–289, 2012.

[12] G. M. Nelson, R. D. Quinn, R. J. Bachmann, W. C. Flannigan, R. E. Ritzmann, and J. T. Watson, "Design and simulation of a cockroach-like hexapod robot," in *Proc. IEEE Robot. Autom., Int. Conf.*, 1997, vol. 2, pp. 1106–1111.

[13] Y. Jiang, H. Zhang, Z. Li, and Y. Deng, "Design and optimization of multiclocked embedded systems using formal techniques," *IEEE Trans. Ind. Electron.*, vol. 62, no. 2, pp. 1270–1278, Feb. 2015.

[14] Y. Jiang *et al.*, "Bayesian-network-based reliability analysis of PLC systems," *IEEE Trans. Ind. Electron.*, vol. 60, no. 11, pp. 5325–5336, Nov. 2013.

[15] R. Wang *et al.*, "Timed automata based motion planning for a self-assembly robot system," in *Proc. IEEE Robot. Autom. Int. Conf.*, 2014, pp. 5624–5629.

[16] L. Li, Z. Shi, Y. Guan, C. Zhao, J. Zhang, and H. Wei, "Formal verification of a collision-free algorithm of dual-arm robot in hol4," in *Proc. IEEE Robot. Autom, Int. Conf.*, 2014, pp. 1380–1385.

[17] S. Ma, Z. Shi, Z. Shao, Y. Guan, L. Li, and Y. Li, "Higher-order logic formalization of conformal geometric algebra and its application in verifying a robotic manipulation algorithm," *Adv. Appl. Clifford Algebras*, vol. 26, no. 4, pp. 1305–1330, 2016.

[18] M. Leucker and C. Schallhart, "A brief account of runtime verification," *J. Logic Algebraic Program.*, vol. 78, no. 5, pp. 293–303, 2009.

[19] Y. Jiang *et al.*, "Use runtime verification to improve the quality of medical care practice," in *Proc. IEEE Softw. Eng. Companion, ACM Int. Conf.*, 2016, pp. 112–121.

[20] Y. Jiang, H. Song, R. Wang, M. Gu, J. Sun, and L. Sha, "Data-centered runtime verification of wireless medical cyber-physical system," *IEEE Trans. Ind. Informat.*, vol. 13, no. 4, pp. 1900–1909, Aug. 2017.

[21] S. Zhang, F. He, and M. Gu, "Verv: A temporal and data-concerned verification framework for the vehicle bus systems," in *Proc. Comput. Commun., Conf.*, 2015, pp. 1167–1175.

[22] S. Adam, M. Larsen, K. Jensen, and U. P. Schultz, "Rule-based dynamic safety monitoring for mobile robots," *J. Softw. Eng. Robot.*, vol. 7, no. 1, pp. 120–141, 2016.

[23] J. Huang *et al.*, "Rosrv: Runtime verification for robots," in *Runtime Verification*, Berlin, Germany: Springer, 2014, pp. 247–254.

[24] C. A. N. Bosch, *Specification version 2.0.* Wrttemberg, Germany: Bosch GmbH, 1991.

[25] H. J. Chen, B. T. Gao, X. H. Zhang, and Z. Q. Deng, "Drive control system for pipeline crawl robot based on can bus," in *Journal of Physics: Conference Series*, vol. 48, page 1233. Bristol, U.K.: IOP Publishing, 2006.

[26] M. Bonani *et al.*, "The marxbot, a miniature mobile robot opening new perspectives for the collective-robotic research," in *Proc. Intell. Robots Syst., 2010 IEEE/RSJ Int. Conf.*, 2010, pp. 4187–4193.

[27] G. Metta, G. Sandini, D. Vernon, L. Natale, and F. Nori, "The icub humanoid robot: an open platform for research in embodied cognition," in *Proc. 8th WorkshopPerform. Metrics Intell. Syst.*, ACM, 2008, pp. 50–56.

[28] A. Bauer, M. Leucker, and C. Schallhart, "Runtime Verification for LTL and TLTL," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, 2011, Art. no. 14.

[29] C. Fritz, "Constructing bchi automata from linear temporal logic using simulation relations for alternating bchi automata," *Lecture Notes in Computer Science*, vol. 2759, pp. 35–48, 2003.

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.