Received March 23, 2018, accepted April 21, 2018, date of publication May 4, 2018, date of current version June 5, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2833213

Modeling and Analysis of the 1-Wire Communication Protocol Using Timed Colored Petri Nets

MARÍA EMILIA CAMBRONERO, HERMENEGILDA MACIÀ, VALENTÍN VALERO, AND LUIS OROZCO-BARBOSA, (Member, IEEE)
Department of Computer Systems, University of Castilla-La Mancha, 02071 Albacete, Spain

Corresponding author: María Emilia Cambronero (memilia.cambronero@uclm.es)

This work was supported in part by the Spanish Ministry of Science and Innovation (cofinanced by European Union FEDER Funds) Project DArDOS (formal development analysis of complex systems in distributed contexts: foundations, tools, and applications) under Grant TIN2015-65845-C3 subproject 2-R and Grant TIN2015-66972-C5 subproject 2-R.

ABSTRACT The widespread use of sensor networks has enabled the deployment of a wide variety of services. In order to reduce maintenance costs without jeopardizing the reliability of the services, developers and researchers are exploring ways to reduce the complexity of the protocols and their underlying hardware infrastructure. However, the verification and evaluation of novel solutions must also be taken as a key design parameter in the development of reliable and cost-effective solutions. In this paper, we develop a timed colored Petri net (TCPN) model of the 1-wire protocol, which is one of the most popular and simplest protocols currently used in the implementation of sensor platforms. The use of TCPNs is justified by the fact that they provide us with the means to evaluate the qualitative and quantitative properties of the protocol. Our results include the analysis of the absence of deadlocks as well as the quantitative evaluation of the protocol. Our numerical results are also confirmed by using an event-driven simulator.

INDEX TERMS Sensors, 1-wire, formal modeling, colored Petri nets, analysis.

I. INTRODUCTION

Nowadays, data communications between devices and sensorized systems and processes are being introduced in numerous sectors, including those of leisure, industry and marketing, among others. In this context, sensor networks are being introduced in the industry, business and social activities. As the number of sensors per network is steadily increasing, efforts are being made to simplify their interconnection while ensuring the quality-of-service requirements of a wide variety of applications. The design of protocols capable of providing both communication and power is nowadays one of the main objectives in the development of self-configurable and reliable sensor platforms. Despite all the efforts made up till now, there are still many open issues when it comes to ensuring the correct auto-configuration and operation of current commercial protocols. Therefore, conducting a rigorous analysis of existing and up-coming protocols is an important step towards guaranteeing their reliability and robustness.

In this paper, we conduct an analysis of the 1-wire communication protocol using Timed Colored Petri Nets (TCPNs) [13]. The 1-wire protocol is a sensor serial communication protocol based on a single wire [18]. The main advantages of this protocol are its simplicity and self configuring features which enable the connectivity of low-cost sensor devices via a single wire plus ground reference to accomplish both communication and power provisioning capabilities. This feature makes the 1-wire protocol an ideal solution for remote applications in which space restrictions or accessibility difficulties make it impractical to deploy dedicated power supplies. A single bus master can support multiple slaves (sensors) over a single low-cost cable bus based on a PC or micro-controller communicating digitally over the cable using 1-wire compliant components.

Bearing in mind that many sensor applications rely on data delivery services, it is important to verify the correct operation of the underlying communications protocols. Among the numerous verification methodologies available nowadays, Petri Nets (PNs) have been successfully used in the analysis of concurrent systems including communications protocols [14]. PNs models are specified by means of a graphical and mathematical modeling language that enables the creation of the *state space*, and more specifically the *state graph* of the system being studied. Furthermore, numerous computer-based tools are now available to researchers and practitioners interested in the specification, verification and performance evaluation of novel protocols [23]. Unfortunately, in numerous cases the state space cannot be constructed, due to the size of the state graph representing all possible states and transitions, thus making the processing of all reachable states intractable. In these cases, and in general, simulation studies should be conducted to supplement the system verification results. In other words, properties, such as, deadlock freeness, state reachability or liveness are verified by using state space methods, while quantitative metrics, such as delays, throughput and performance indices can only be obtained via simulations.

Petri nets have been extended in different ways in order to enrich their power of description. In particular, Timed Colored Petri Nets (TCPNs) are an extension of Petri nets that is capable of modeling variables and times associated with them, making this a very suitable formal method for the analysis of sensor communication networks. TCPNs have been introduced to represent large complex systems, in which their components can be encoded as colors attached to the tokens in the places. Thus, TCPNs allow us to create compact and parameterized representations of component-based systems, without losing the analysis capabilities of standard PNs. Timed Colored Petri nets are supported by *CPN Tools* [24], which is a well-known tool for the analysis and simulation of Petri nets. Thus, the analysis of the 1-wire protocol can benefit from the use of TCPNs, since they can formally describe almost all the crucial aspects of the protocol, specifically timed aspects, data and protocol commands. In addition, the use of TCPNs allows us to have either manual or automatic simulations of the protocol behavior by changing the system configuration, for instance, modifying the number of sensors.

In this work, we develop a TCPN model of the 1-wire protocol, which is a technology that is widely used in the implementation of sensor systems. We make use of the main TCPNs features, which allow us to build a system consisting of a large number of sensors. The evaluation comprises the analysis of the space state and numerical performance metrics. We numerically evaluate the protocol using CPN Tools.

The rest of the paper is organized as follows. Section II presents the related work. Section III reviews the basic principles and notation of TCPNs, and describes the operation of the 1-wire protocol. Section IV describes our TCPN model for the 1-wire sensor communication protocol. We highlight the main benefits in terms of scalability and readability when applying the hierarchical and organizational features of TCPNs in the modeling process. Section V and Section VI report our protocol analysis results and performance evaluation results, respectively. Finally, Section VII contains our conclusions and a list of our future work plans.

II. RELATED WORK

Colored Petri Nets (CPNs) have successfully been used in the analysis and evaluation of numerous communication networks and protocols. Hu and Jiao [10] have analyzed the IEEE802.15.4 wireless protocol by exploiting the hierarchical features of CPN Tools. They develop a compact and scalable CPN model by using hierarchical and symmetrical modeling techniques. They simulate their CPN model in terms of throughput, delivery ratio, delay, and energy cost. They validate their results by comparing them with the ones obtained using the well-known *ns*-2 simulation tool. Azgomi and Khalili [1] present a model of an energy-aware MAC protocol using CPNs to evaluate the power consumed by the nodes of a wireless sensor network. Ben-Othman et al. [3] have also studied the performance evaluation of the EQ-MAC protocol by using CPNs. They have shown the benefits of using CPNs in the modeling and evaluation of two relevant metrics, namely average delay and packet delivery ratio.

Zhang et al. [28] use CPNs to model the interaction between sensor devices and the geographic environment. They classify the IoT (Internet of Things) service, geographic entity and GIS (Geographic Information Systems) services as data and processing services. To this end, they use an algebra and CPNs to model and analyze geo features, IoT and GIS services, and the interaction process between the sensor network and the geographic environment. The authors demonstrate the adequacy of the CPNs-based evaluation methodology via three different case studies.

Zairi et al. [26] propose a CPN approach to model the global behavior of wireless sensors networks including the network energy consumption. Their approach is based on the concept of components oriented modeling and the expressiveness of CPNs. The focus is on two components: the radio system and the MAC protocol. The authors then construct the model of each subnet and interface separately. They evaluate the network in terms of the energy consumption, thus allowing them to predict the network lifetime.

Macià et al. [15] present the Network rOle-based Routing Intelligent Algorithm (NORIA) protocol model and perform evaluation using Prioritized-Timed Colored Petri Nets (PTCPNs). The main purpose of this algorithm is to reduce energy consumption and improve data routes. The authors present a state space analysis and performance evaluation in order to prove that the model is deadlock free.

Vanit-Anunchia et al. [25] have used CPNs to analyze the Datagram Congestion Control Protocol (DCCP). They identify the state space explosion as one of the major challenges to overcome in the analysis of finite state systems. They then propose the use of the sweep-line method, which is a state exploration method based on the notion of progress to allow states to be deleted from memory when they are no longer required. Billington and Vanit-Anunchia [4] propose the use of incremental enhancements as an alternative approach towards the development of CPN models of more complex protocols, and Billington and Yuan [5] undertake the development of a CPN model of the MANET routing protocol. In their own words, the careful crafting of a CPN model should result in smaller state spaces, i.e., tractable CPN models.

Muzaffar et al. [19] present an algorithm for detecting and setting the Pulsed-Index Communication (PIC) protocol parameters. The protocol allows the master device to configure all the slave devices connected to a channel network. In a similar way to the 1-wire protocol, this protocol operates over a single-channel as a means to simplify the communication protocol operation. Dudak and Cicak [7] have developed a CPN model of the MODBUS protocol. Like the 1-wire protocol, the MODBUS protocol defines the master/slave communication rules over a serial line. Zhan et al. [27] verify the operation of the master/slave polling mechanism of the vehicle protocol IEC-61375. By using a CPN model, they have been able to detect a defect in the coupling of an FPGA and an ARM system. Their results show the benefits of using CPNs in the formal verification of communications protocols.

CPNs have also been applied to the analysis of event flows taken from a sensor network [16], where Complex Event Processing (CEP) technologies are used to represent critical situations derived from event flows. These CEP models are then transformed into equivalent CPN representations so as to apply the verification and validation techniques mentioned above. Other extensions of PNs have also been considered for the analysis of communication protocols. Stochastic Petri nets (SPNs) are an extension in which transitions have a negative exponential distribution associated with them in order to establish a random delay in their firing. Heindl and German [9] have used SPNs for the performance evaluation of the IEEE 802.11 wireless LANs protocol.

From the above discussion, it should be clear that the modeling of even simple network protocols consisting of a large number of components is a challenging task. It is also important to realize that the validation and modeling of network protocols, such as the 1-wire protocol, are key to the deployment of a large number of services. Many of the emerging end-user applications will be characterized by their stringent requirement in terms of reliability, throughput and delays.

To the best of the authors' knowledge, there are very few works on the analysis and evaluation of the 1-wire protocol [17], [18], [22], and none of them have made use of formal validation techniques. Chew et al. [6] present a low-cost temperature sensors-data loggers, called Thermochron iButtons, which are based on 1-wire data communications. The experimental results are validated by experimental temperature monitoring of a power transformer of one of the residential area substations during 24 hours. Lei et al. [2] focus on reading data errors on a specific 1-wire bus digital temperature sensor, specifically the DS18B20. For this purpose, they describe a testing error method. Another work based on 1-wire is presented by Gosheblagh and Mohammadi [8], in which the authors propose a monitoring system based on the 1-wire protocol, in order to meet the reliability requirements of the sensor networking and bus controller. Perera et al. [20] propose a single sensor node solution to interconnect transducers to sensor networks using Field-Programmable Gate Array (FPGA). Their system consists of sensor devices placed at different geographical locations and controlled from a single central control site. The authors claim that by integrating all the control mechanisms in a single core using the 1-wire protocol, the processing power speedup can be considerably increased. However, they do not validate the operation or evaluate the performance of their proposal, which is essential for many sensitive applications.

III. BACKGROUND

A. TIMED COLORED PETRI NETS

A Petri Net (PN) is a directed bi-partite graph with nodes of two types: places (drawn as circles) and transitions (drawn as rectangles). An arc can connect either a place with a transition (pt-arc) or a transition with a place (tp-arc). Let P be the set of places, T the set of transitions, $X = P \cup T$ (nodes) and $F \subseteq (P \times T) \cup (T \times P)$ the set of arcs. For any node $x \in X$ (place or transition), we define the preconditions and postconditions of x, denoted by ${}^\bullet x$ and $x^\bullet$ respectively, as follows: ${}^\bullet x = \{y \in X | (y, x) \in F\}, x^\bullet = \{y \in X | (x, y) \in F\}$.

Places usually represent states or system conditions, while transitions are the actions or events that produce changes in the system state. Places can have an associated *marking*, which is a natural number indicated beside the place (number of *tokens* on it). This number can be used, for instance, to indicate the number of packets or the number of processes in a queue or some other system load conditions.

Pt-arcs are also labeled with a natural number (*arc weight*) to indicate the number of tokens required to execute (*fire*) the outgoing transition. The default value is one, in which case there is no need to explicitly indicate it. The same notation applies to tp-arcs, but in this case the weight indicates the number of tokens to be produced at the outgoing place when the transition is fired. Thus, for a transition to be fireable (*enabling condition*), all its precondition places must have at least as many tokens as the weight of the arc that connects them to the postcondition transition.

The firing of a transition t has therefore the following effects:

- For each precondition place $p \in {}^\bullet t$, a number of tokens equal to the weight of the pt-arc (p, t) are removed from p.
- For each postcondition place $p \in t^\bullet$, a number of tokens equal to the weight of the tp-arc (t, p) are produced at p.

In the simple model, no time information is considered and no information can be associated to the tokens or places, these being two important features required to model concurrent systems. Colored PN (CPN) [13] is an extension to the original Petri nets which incorporates data and time, making it possible to model complex data structures attached to tokens and time restrictions in the sequence and synchronization of the processes involved. Thus, in CPNs, places have an associated *color set* (a data type), which specifies the set of permitted token colors at a given place. CPNs are supported by a widely used tool, namely CPN Tools [24], which allows us to create, edit, simulate and analyze CPNs. The notation described below is the one used in this tool.

A place can have no attached information at all, as in the plain model. In this case, we indicate *UNIT* as the color set of the place. However, as a color set, a place can now have, for instance, the set of integer numbers *INT*, a Cartesian product of two or more color sets such as $INT2 = INT \times INT$, a string (*STRING*), etc. In this case, each token has an attached data value *(color)*, which belongs to the corresponding place color set. Furthermore, we can use the timed features of CPNs. In this type of nets, a discrete global clock is used to represent the total time elapsed in the system model; and places can be either timed or untimed. In the case of timed places, their tokens have an associated timestamp, which indicates the time at which they will be available and thus usable to fire a transition. In CPN Tools, the current number of tokens at every place is drawn in green on the right-hand side of the place circle, and the specific colors of these tokens are indicated by the notation n'v, meaning that there are n instances of color v. The symbol '++' (or '+ + +' for timed tokens) is used to represent the union of colors in CPN Tools. Thus, a timed integer place (color set *int timed*) with a marking $2'3@5 + + + 1'9@10$ has 2 tokens with value 3 and timestamp 5 and 1 token with value 9 and timestamp 10.

The arc inscriptions are now extended to color set expressions, which are constructed using variables, constants, operators and functions. The arc expressions must evaluate a color or multiset of colors in the *color set* of the attached place. Tp-arcs[1] can have a delay associated, with the syntax $n'v@x$ to denote that n tokens with value v are produced with a timestamp equal to the current time increased by x time units. Delays can also be indicated in transitions, with the syntax $@+x$, which means that all tokens produced will have as the current time plus x timestamp. Furthermore, transitions can have guards that can restrict their firing, as well as priorities. Guards are Boolean expressions that constructed by using the variables, constants, operators and functions of the model, and they must be evaluated as true for the transition to be *fireable*. Transitions can also have an associated priority, so in the event of a conflict between two transitions that can be fired at a given time, the transition with the highest level of priority is fired first.

For any transition t with variables $x1, x2, \ldots$ in its input and output arc expressions, we call a *binding* of t an assignment of concrete values to each of these variables. A binding of a transition t is then *enabled* if there are tokens in its precondition places matching the values of the corresponding inscriptions. Thus, arc expressions are evaluated by assigning values to the variables, and these values are then used to select the tokens that must be removed or added when firing the corresponding transition. When no transition can be fired at the current time, time elapsing occurs, but only up to a time at which some transition can be fired.

[1]Pt-arcs can also have delays associated with them, but this feature is not used in this paper.

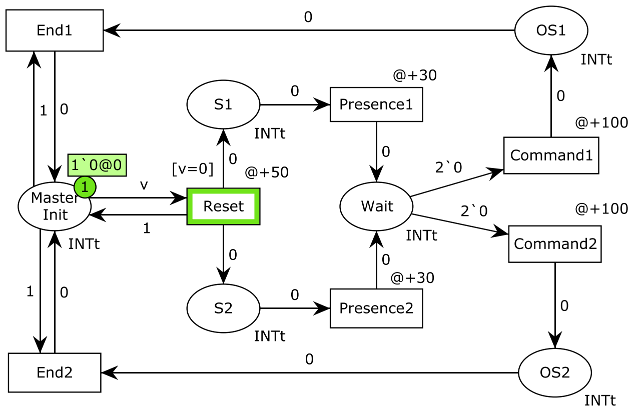

FIGURE 1. **Graphical view of a CPN.**

Example 1: Figure 1 shows a TCPN modeling a simple master-slaves protocol. In this TCPN, the color set *INTt* stands for timed integer (*int timed* in CPN Tools) and v is an integer variable. Place *MasterInit* has initially one token with value 0, available at time 0. Transition *Reset* is depicted in green, which means that there is an enabled binding for it ($v = 0$). The firing of *Reset* changes the token value at *MasterInit* to 1 and writes one 0-valued token at both $S1$ and $S2$, which will be available at time 50. At this time, both *Presence* transitions must be fired, thus producing two 0-valued tokens at *Wait* with timestamp 80. Then, either *Command1* or *Command2* is fired at time 80, after which the corresponding *End* transition is fired at time 180 to change the token value at *MasterInit* to 0 again. □

The TCPN in Figure 1 represents a very simple system. In general, we will have to deal with larger TCPNs where the use of the hierarchical features of CPN Tools will enable to the decomposition of the model into various smaller subnets. These subnets or *pages* in CPN Tools terminology, should be linked by using substitution transitions and fusion sets. Substitution transitions refer to transitions replaced by subnets represented on other pages, while fusion sets are sets of places used on different pages, which are functionally identical and therefore correspond to the same place from a formal viewpoint. In this paper, we have made use of fusion places to split the model into two pages. The link between the two pages is made by means of their common places, denoted by a blue fusion label in their bottom left-hand corner.

B. 1-WIRE PROTOCOL

The 1-wire communication protocol is an asynchronous serial protocol for communication based on a single data line [17], [18], [22], which follows a master/slave scheme. A master must always be connected to the bus serving one or more slave devices. The communication is half-duplex and signaling on the bus is divided into time slots. Slave devices are allowed to have a time base that may significantly differ from the nominal time base. This requires the master to take full responsibility for maintaining communication synchronization with the slaves, and so all data exchange over the bus is performed under the command of the master.

Every communication on the bus must be initiated by the master. The *reset* command is used to set all the devices in a known state. Two other sets of commands are defined on the bus: *ROM-function commands* and *memory-function commands*. The former set defines the means to manage the slave addresses and alarm events. The latter set includes the read and write commands to be performed by the master on the internal memory of the slaves.

1) ***Reset/Wait/Presence***
The main function of the reset command is to prepare all the slave devices for a command. The master drives the bus low for eight time slots (between 480 and 640μs) to reset all the slaves. The master then releases the bus by pulling it up. In response to this command, the slaves connected to the bus pull the bus low to indicate their presence. In order to ensure a proper synchronization of the system, the slaves should not pull down the bus before 15μs and not later than 60μs, counting from the beginning of the pull up bus operation performed by the master. We will use the term *wait* to refer to this latter $15-60\mu$s time period elapsed from the beginning of the pull-down performed by the slaves, see Figure 2. The master should then sample the bus at one point in time during the 240μs following the wait period. It is obvious that in the absence of slaves, the master will become aware of this fact by detecting a high-level bus condition. The 1-wire communication always starts with the *reset/wait/presence* sequence.

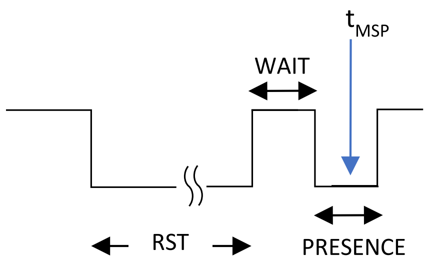

FIGURE 2. Reset/Wait/Presence cycle.

2) ***ROM Function Commands***
A globally unique 64-bit identifier number identifies every 1-wire device, which is stored in ROM. The slaves on the bus are addressed by this number. This identifier consists of three parts: an 8-bit family code, a 48-bit serial number, and an 8-bit CRC computed from the first 56 bits. For simplicity, we use integers to define the slave identifiers in our model. Therefore, some commands that operate on the 64-bit identifier are defined, and called ROM function commands. In this paper, we use the following ROM commands:

- Match ROM. This is used on a bus with multiple slave devices to address an individual slave device. The *Match ROM command* is transmitted on the bus followed by the complete 64-bit slave identifier for the device selected.
- Alarm Search ROM. This identifies and addresses the devices whose temperature is outside the programmed alarm limits[2] [12]. By using this command the master device can determine whether any device experienced an alarm condition during the most recent temperature conversion. After every Alarm Search cycle (i.e., Alarm Search command followed by data exchange), the bus master must return to the initialization step, that is, a reset signal is expected.

3) ***Memory Function Commands***
The memory function commands [11] are addressed to a specific slave device, and they usually deal with writing or reading operations to/from the internal memory and registers on slave devices. In general, each device will have its own memory function commands, so we will consider the following three commands: 1) the unstructured read/write operations of the device memory; 2) the use of a packet structure called the Universal Data Packet (UDP); and 3) a combination of multiple UDP structures into a file structure. In this paper, we will only consider the first type, namely unstructured read and write for sensor memories. The *Read* command is used to retrieve the temperature measurement values from sensor memories, while the *write* command is used to reprogram the alarm temperature limits in their scratchpad memories [11].

IV. TIMED COLORED PETRI NET MODEL FOR 1-WIRE

In this section, we describe the TCPN modeling of the main commands of the 1-wire sensor communication protocol. The model consists of one master and several slaves[3] (sensors) connected to the bus. The TCPN has been structured in two parts (pages) using the hierarchy capabilities of CPN Tools. The first page is used to implement the TCPN model of the master and the second one for the TCPN model of the slaves. We have linked the two different pages using fusion places, which are a TCPN facility enabling the use of the same place on several pages. It is worth mentioning that the model can be easily reconfigured, e.g. by simply changing the value of a constant, such as the number of sensors, *ns*, which means that a model of different dimensions can be easily implemented, simulated and evaluated.

Figures 3 and 4 show the master and slave TCPN models for a system consisting of a master and five slaves. The main elements of the Master TCPN are shown in Figure 3. The *Master* and *Slaves* places model the master and slave

[2]In this paper, we mainly consider the use of temperature sensor devices.
[3]We use the term sensor or slave equally to refer to the sensors connected to the bus.

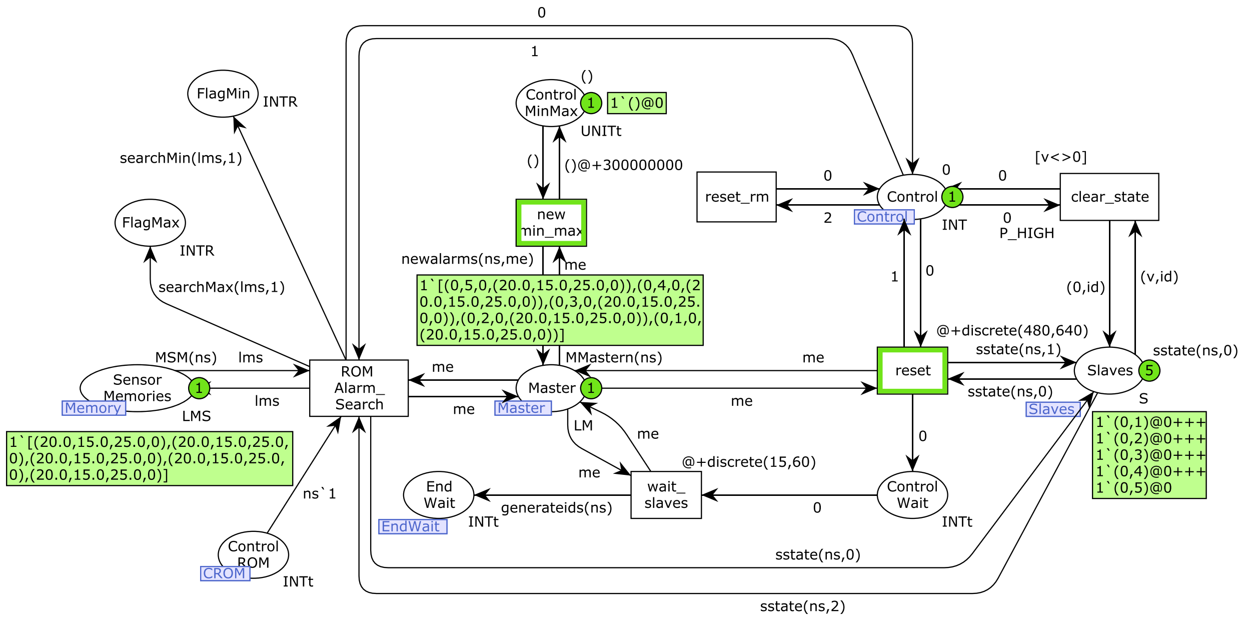

FIGURE 3. Master TCPN.

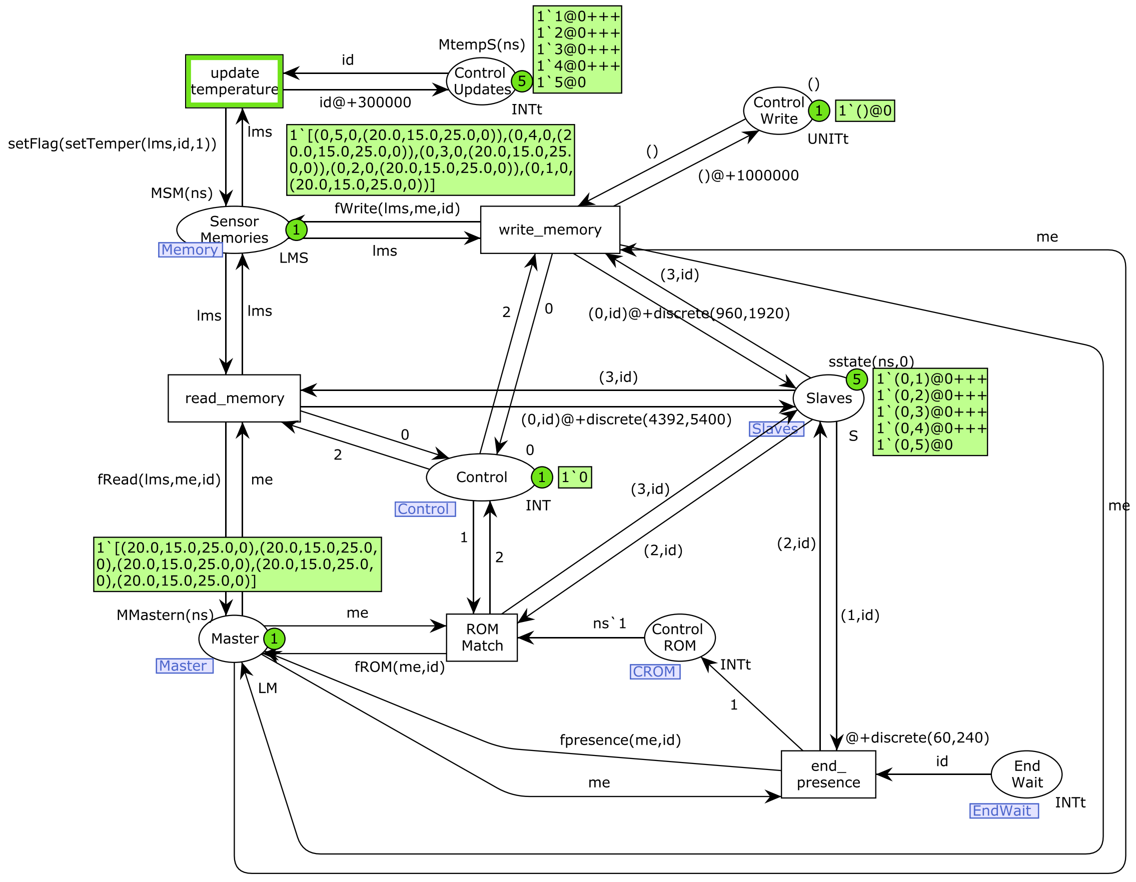

FIGURE 4. Slave TCPN.

states, respectively. The *Control* place is used to model the *reset*, *read_memory* and *write_memory* control operations.

The *reset* and *clear_state* transitions are used to initialize the 1-wire protocol. The *wait_slaves* transition models the

TABLE 1. Colset information.

Colset Type	Used by	Colset Component Type	Component Meaning
B	*Slaves*	Int [0..1]	Boolean Value Stored
D	*Sensor Memories*	Int[0..2]	0: Alarm not triggered. 1: Minimum temperature alarm triggered. 2: Maximum temperature alarm triggered.
C	*Slaves*	Int[0..3]	0: Initial Slave state. 1: Reset signal received. 2: Presence pulse sent. 3: Memory command performed.
S	*Slaves*	C*INT	C: Slave state.
			INT: Slave Id.
MS	*Sensor Memories*	REAL*REAL*REAL*D	REAL: Measured Temperature.
			REAL: Minimum Alarm Threshold.
			REAL: Maximum Alarm Threshold.
			D: Flag Value.
LMS	*Sensor Memories*	List of MS	
M	*Master*	INT*INT*B*MS	INT:Presence Signal Received.
			INT: Slave Id.
			B: Rom Command sent or not.
			MS: Sensor Memory Information.
LM	*Master*	List of M	
INTR	*Master*	INT*REAL	INT: Slave Id. REAL: Temperature Value.

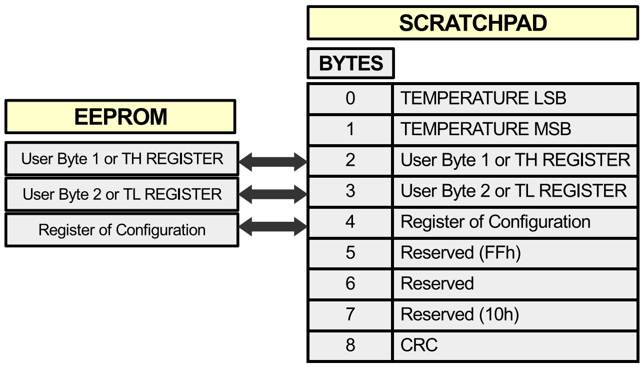
SCRATCHPAD
BYTES
EEPROM
User Byte 1 or TH REGISTER
User Byte 2 or TL REGISTER
Register of Configuration
0 TEMPERATURE LSB
1 TEMPERATURE MSB
2 User Byte 1 or TH REGISTER
3 User Byte 2 or TL REGISTER
4 Register of Configuration
5 Reserved (FFh)
6 Reserved
7 Reserved (10h)
8 CRC

FIGURE 5. Sensor memory structure.

delay following the reset signal sent by the master. The *new_min_max* transition generates the new minimum and maximum alarm threshold values. The *ROM_Alarm_Search* transition models the corresponding ROM command in order to obtain the alarms generated.

The main elements of the slave TCPN are shown in Figure 4. The *Master* and *Slaves* places (fusion places) correspond to the same ones included on the Master TCPN page. The *end_presence* transition models the presence pulse sent by the slave. The *ROM Match* transition models the Match ROM performed by the Master. The *Sensor_Memories* place (a fusion place) contains the sensor temperature information. The *write_memory* and *read_memory* transitions correspond to the write and read operation performed by the Master following a Match ROM command. Table 1 shows the sets of colors (colsets) used on both TCPN pages, with the information they capture. Appendix describes in detail the constants, variables and functions used in the model (see Table 7).

As shown in Figure 3, the Master initial marking is set by the *MMastern(ns)* function, which returns one token whose color is a list of *ns* M-elements. Element *i* of the list stores the initial information for sensor *i*: (0, *i*, 0, (20.0, 15.0, 25.0, 0)) (see M in Table 1). Analogously, function *sstate(ns,0)* prepares the initial marking at place *Slaves*, in this case *ns* S-tokens. Each token carries the state information of one sensor. The initial marking for sensor *i* is then (0, *i*) (see S in Table 1). Place *Sensor_Memories* is also initialized with one token, whose color is a list with the memory information of each sensor (measured temperature, minimum alarm threshold, maximum alarm threshold and alarm flag).

The master initializes the communication by sending a reset signal, modeled by the firing of transition *reset*. Its firing does not change the marking of the *Master*, but changes the token value at place *Control* from 0 to 1, inhibiting a new firing of *reset*. It also writes one token at place *Control_Wait* and updates all sensor states, writing 1 in the first field of all of them (function *sstate(ns,1)*). Furthermore, transition *reset* has an associated delay modeled by a *discrete* function, which sets the timestamp of the token produced at *Control_Wait* with a value in the interval [480, 640]. Once this time has elapsed, transition *wait_slaves* fires, writing *ns* integer tokens at place *End_Wait* (function *generateids(ns)*), each one associated with a sensor identifier, from 1 to *ns*. Transition *end_presence* is then fired *ns* times (Figure 4), thus signaling the presence of the sensors to the Master. Sensor states are also updated by the firing of transition *end_presence*, which changes their state fields from 1 to 2. In addition, each firing updates the corresponding sensor presence information at place Master (function *fpresence*) and writes one token at place *Control_ROM* with a timestamp in the interval [60, 240].

Once all the presence pulses are received, a ROM command can be executed, and in our case we only consider either a ROM Match or a ROM Alarm Search. Thus, once we have *ns* tokens at *Control_ROM* and all of them are available, both transitions *ROM_Match* (Figure 4) and *ROM_Alarm_Search* (Figure 3) are enabled, but only one of them can be fired, as they both remove the *ns* tokens from place *Control_ROM*.

Transition *ROM_Match* selects a sensor in order to perform a memory command on it, something which is nondeterministically chosen in the model. In a manual simulation, we can set the binding of the variable *id* to a given sensor. In an automatic simulation setup, the CPN Tools simulator engine will arbitrarily choose any of them. Its firing changes the state of sensor *id* from 2 to 3, and it also changes the value of the token at place *Control* from 1 to 2 (which activates the

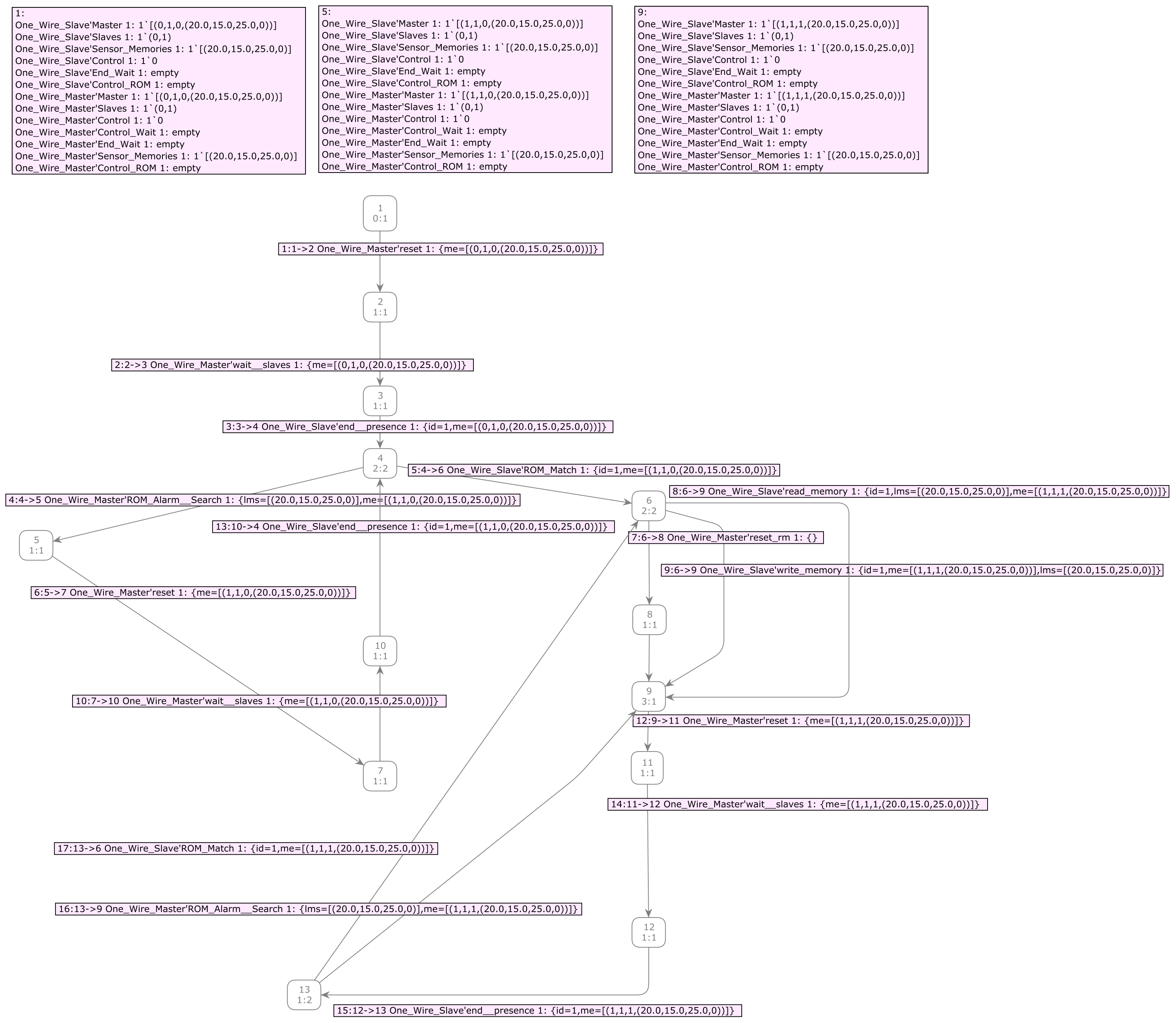
1:
One_Wire_Slave'Master 1: 1`[(0,1,0,(20.0,15.0,25.0,0))]
One_Wire_Slave'Slaves 1: 1`(0,1)
One_Wire_Slave'Sensor_Memories 1: 1`[(20.0,15.0,25.0,0)]
One_Wire_Slave'Control 1: 1`0
One_Wire_Slave'End_Wait 1: empty
One_Wire_Slave'Control_ROM 1: empty
One_Wire_Master'Master 1: 1`[(0,1,0,(20.0,15.0,25.0,0))]
One_Wire_Master'Slaves 1: 1`(0,1)
One_Wire_Master'Control 1: 1`0
One_Wire_Master'Control_Wait 1: empty
One_Wire_Master'End_Wait 1: empty
One_Wire_Master'Sensor_Memories 1: 1`[(20.0,15.0,25.0,0)]
One_Wire_Master'Control_ROM 1: empty
5:
One_Wire_Slave'Master 1: 1`[(1,1,0,(20.0,15.0,25.0,0))]
One_Wire_Slave'Slaves 1: 1`(0,1)
One_Wire_Slave'Sensor_Memories 1: 1`[(20.0,15.0,25.0,0)]
One_Wire_Slave'Control 1: 1`0
One_Wire_Slave'End_Wait 1: empty
One_Wire_Slave'Control_ROM 1: empty
One_Wire_Master'Master 1: 1`[(1,1,0,(20.0,15.0,25.0,0))]
One_Wire_Master'Slaves 1: 1`(0,1)
One_Wire_Master'Control 1: 1`0
One_Wire_Master'Control_Wait 1: empty
One_Wire_Master'End_Wait 1: empty
One_Wire_Master'Sensor_Memories 1: 1`[(20.0,15.0,25.0,0)]
One_Wire_Master'Control_ROM 1: empty
9:
One_Wire_Slave'Master 1: 1`[(1,1,1,(20.0,15.0,25.0,0))]
One_Wire_Slave'Slaves 1: 1`(0,1)
One_Wire_Slave'Sensor_Memories 1: 1`[(20.0,15.0,25.0,0)]
One_Wire_Slave'Control 1: 1`0
One_Wire_Slave'End_Wait 1: empty
One_Wire_Slave'Control_ROM 1: empty
One_Wire_Master'Master 1: 1`[(1,1,1,(20.0,15.0,25.0,0))]
One_Wire_Master'Slaves 1: 1`(0,1)
One_Wire_Master'Control 1: 1`0
One_Wire_Master'Control_Wait 1: empty
One_Wire_Master'End_Wait 1: empty
One_Wire_Master'Sensor_Memories 1: 1`[(20.0,15.0,25.0,0)]
One_Wire_Master'Control_ROM 1: empty
1 0:1
1:1->2 One_Wire_Master'reset 1: {me=[(0,1,0,(20.0,15.0,25.0,0))]}
2 1:1
2:2->3 One_Wire_Master'wait__slaves 1: {me=[(0,1,0,(20.0,15.0,25.0,0))]}
3 1:1
3:3->4 One_Wire_Slave'end__presence 1: {id=1,me=[(0,1,0,(20.0,15.0,25.0,0))]}
4 2:2
5:4->6 One_Wire_Slave'ROM_Match 1: {id=1,me=[(1,1,0,(20.0,15.0,25.0,0))]}
4:4->5 One_Wire_Master'ROM_Alarm__Search 1: {lms=[(20.0,15.0,25.0,0)],me=[(1,1,0,(20.0,15.0,25.0,0))]}
6 2:2
8:6->9 One_Wire_Slave'read_memory 1: {id=1,lms=[(20.0,15.0,25.0,0)],me=[(1,1,1,(20.0,15.0,25.0,0))]}
13:10->4 One_Wire_Slave'end__presence 1: {id=1,me=[(1,1,0,(20.0,15.0,25.0,0))]}
5 1:1
7:6->8 One_Wire_Master'reset_rm 1: {}
9:6->9 One_Wire_Slave'write_memory 1: {id=1,me=[(1,1,1,(20.0,15.0,25.0,0))],lms=[(20.0,15.0,25.0,0)]}
6:5->7 One_Wire_Master'reset 1: {me=[(1,1,0,(20.0,15.0,25.0,0))]}
8 1:1
10 1:1
9 3:1
10:7->10 One_Wire_Master'wait__slaves 1: {me=[(1,1,0,(20.0,15.0,25.0,0))]}
12:9->11 One_Wire_Master'reset 1: {me=[(1,1,1,(20.0,15.0,25.0,0))]}
11 1:1
7 1:1
14:11->12 One_Wire_Master'wait__slaves 1: {me=[(1,1,1,(20.0,15.0,25.0,0))]}
17:13->6 One_Wire_Slave'ROM_Match 1: {id=1,me=[(1,1,1,(20.0,15.0,25.0,0))]}
16:13->9 One_Wire_Master'ROM_Alarm__Search 1: {lms=[(20.0,15.0,25.0,0)],me=[(1,1,1,(20.0,15.0,25.0,0))]}
12 1:1
13 1:2
15:12->13 One_Wire_Slave'end__presence 1: {id=1,me=[(1,1,1,(20.0,15.0,25.0,0))]}

FIGURE 6. State graph for one sensor ($ns = 1$).

reset_rm, *read_memory* and *write_memory* transitions) and updates the information stored at place *Master* for sensor *id* to indicate that a ROM command has been performed (function *fROM*).

After *ROM_Match*, either *reset_rm*, *read_memory* or *write_memory* can be fired, but the latter has a time restriction (timed token at place *Control_Write*). *Reset_rm* sets the token at place *Control* to 0, which enables the firing of transition *clear_state* for those sensors that have a non-zero value in their slave state information. This transition has a high priority and thus, it must be fired immediately if enabled. *Clear_state* sets the first field of the slave state to 0. Once all of slave states have 0 in their first field, transition *reset* can be fired again to start a new cycle.

Sensor memories consist of a scratchpad organized in nine bytes (Figure 5), including the two bytes of EEPROM that store the high and low temperature values that trigger the alarms. A write operation writes two bytes in the scratchpad memory, specifically bytes 2 and 3, corresponding to volatile copies of the 1-byte upper and lower alarm trigger registers (TH and TL). A write takes between 960 and 1920μs, since writing a bit takes between 60 and 120μs. Transition *write_memory* modifies the alarm limits at place *Sensor_Memories* for sensor *id* (function *fWrite*), taking the information from the *Master* place. Rewriting the alarm limits is a rather unusual operation, as we do not expect the limits to change frequently. Accordingly, we have decided to restrict its firing by including the *Control_Write* place, which has a timed token that enforces a delay of 1 second between the firings of transition *write_memory*. This is still a very short period for rewriting the limits, but long enough in terms of the values considered in the model to severely affect the execution of other operations. We have decided to include this operation in the model to dimension and illustrate the multi scale capabilities of TCPNs tools. The length of this operation period can be dimensioned to reflect the actual system operation. The writing time is captured by the *discrete* function on the arc to *Slaves*, so the corresponding token for this sensor will only be available after the time returned by this function has elapsed.

Transition *read_memory* will always be activated after *ROM_Match*, which reads the current values into the memory of sensor *id*. A read command reads the complete scratchpad memory (9 bytes) of the chosen sensor. Reading a bit takes a time of between 61 and 75μs, so the time to read nine bytes is between 4392 and 5400μs. Accordingly, transition *read_memory* updates the memory information stored at place *Master* for sensor *id* (function *fRead*). It also modifies the corresponding token at place *Slaves*, changing its state to 0 and aging it by a time in the interval [4392, 5400] with respect to the current model time.

Both transitions *read_memory* and *write_memory* set the token at place *Control* to 0, to enable transition *clear_reset*. *Clear_state* is fired several times to set all the sensor states to 0, after which transition *reset* becomes enabled again.

As mentioned above, transition *ROM_Alarm_Search* can also be fired after all slaves have signaled their presence. Its firing removes the *ns* tokens from place *Control_ROM*, which disables transition *ROM_Match*. *ROM_Alarm_Search* checks the sensor alarm status, taking the information from place *Sensor_Memories* and writing tokens at places *FlagMin* and *FlagMax* when some alarms have been triggered.

The Master TCPN includes the transition *new_min_max* to produce new minimum and maximum temperature alarm thresholds, which are then used in subsequent writing operations. Writing new threshold values should be a rather unusual operation, so we could even avoid these updates. However, as mentioned above, our aim is to develop a flexible model that allows operations spanning different time scales, so we have included a threshold update period of 30*s*. In a similar way, the Slave TCPN includes the transition *update_temperature*, which updates the temperature of one sensor. The period for this operation is 300000μs, so all sensor temperatures are updated every 0.3 seconds. Function *setTemper* is used to compute a new temperature value for one sensor, as a slight modification of its previous value, since temperature values do not change discontinuously. For this purpose, a normal distribution $N(0, 3)$ is used to determine this modification. The firing of *update_temperature* changes the sensor information at place *Sensor_Memories*, as well as

TABLE 2. Space state report - statistics for one sensor.

```
 Statistics
------------------------------
  State Space
     Nodes:  13
     Arcs:   17
     Secs:   0
     Status: Full

Home Properties
------------------------------------
 Home Markings
     6 [6,9,11,12,13,...]

 Liveness Properties
------------------------------------
 Dead Markings
     None

  Dead Transition Instances
     None

  Live Transition Instances
     All
```

TABLE 3. Report space state - liveness properties - ns=5.

```
Statistics
--------------------
  State Space
     Nodes:  3231
     Arcs:   8728
     Secs:   1
     Status: Full

 Home Properties
-----------------------
 Home Markings
     149 [3083,3089,3105,3139,3175,...]

 Liveness Properties
------------------------------
 Dead Markings
     None

  Dead Transition Instances
     None

  Live Transition Instances
     All
```

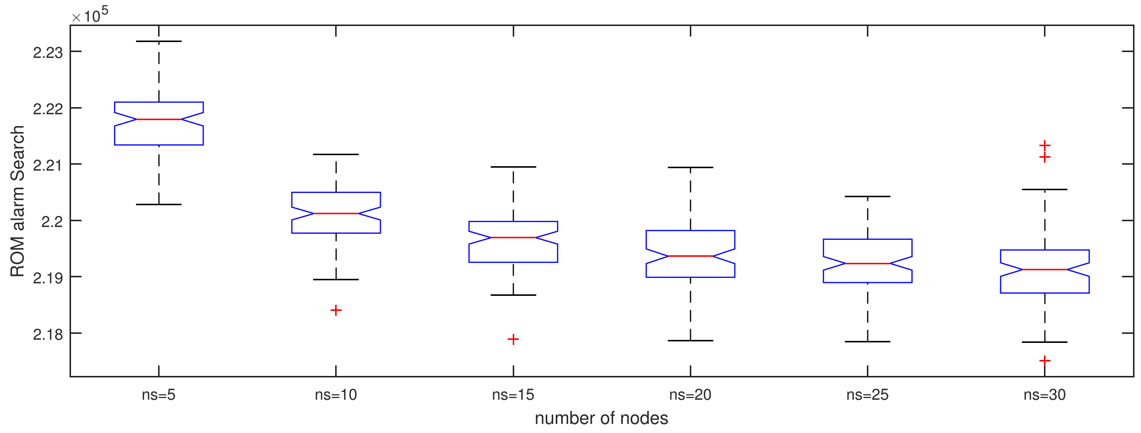

FIGURE 7. ***ROM*** **alarm Search monitor simulations (100 replications).**

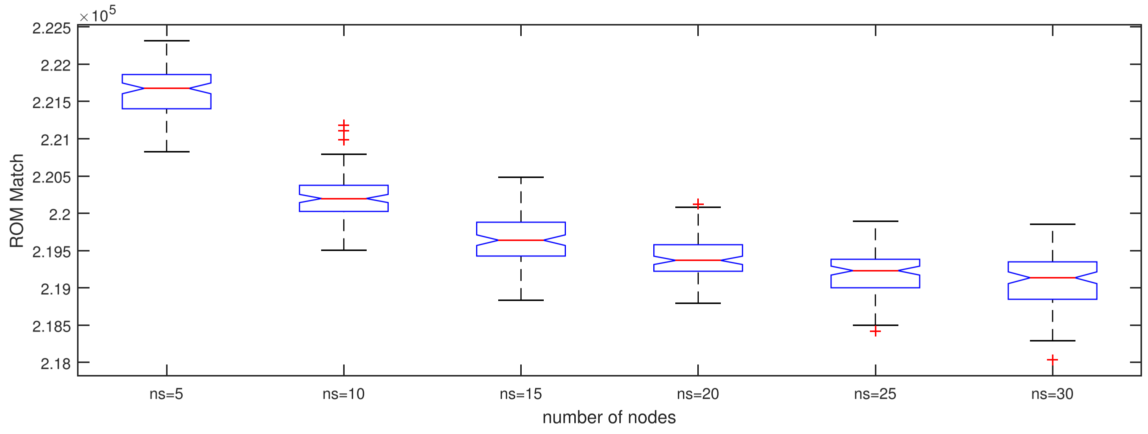

FIGURE 8. ***ROM*** **Match monitor simulations (100 replications).**

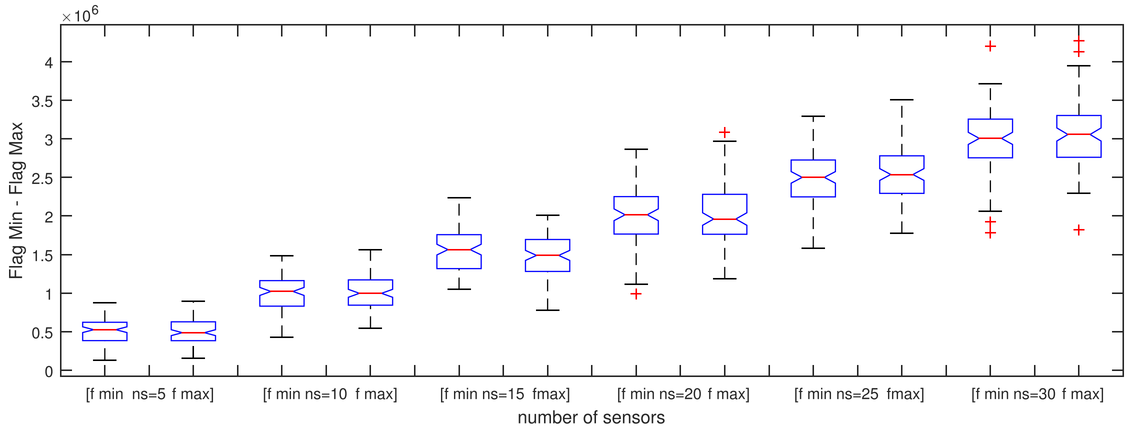

FIGURE 9. ***Flag*** **monitor simulations (100 replications).**

the corresponding alarm flags that could have changed as a consequence of the new temperature value (function *setFlag*).

V. ANALYSIS

In this section, we describe the analysis that has been performed by constructing the state space of the TCPN model. First, we consider the TCPN model without time values, since the network structure depicts the sequence order of the 1-wire commands, and therefore our first objective is a qualitative analysis.

We have first analyzed a simple version of the system with no updates and a single sensor ($ns = 1$). In this

TABLE 4. CPN tools performance report for 100 replications.

	Average	90% Half-Length	Standard Deviation	minimum	maximum
ns=5					
Flag Max	514480.350000	26831.122018	161147.879989	158287	896391
Flag Min	502598.600000	26784.864619	160870.057769	132137	876302
ROM Alarm Search	221749.150000	99.140051	595.435744	220284	223180
ROM Match	221647.820000	57.171717	343.373678	220825	222314
Read	110401.860000	20.319961	122.041806	110124	110663
Write	890.730000	0.074292	0.446196	890	891
ns=10					
Flag Max	1012930.630000	37955.662170	227961.934954	546008	1564285
Flag Min	1010713.070000	37930.513658	227810.892838	427948	1486504
ROM Alarm Search	220108.060000	89.109271	535.190815	218406	221174
ROM Match	220199.580000	51.886362	311.629804	219506	221182
Read	109678.500000	18.028276	08.277935	109348	109992
Write	890.630000	0.080792	0.485237	890	891
ns=15					
Flag Max	1481194.470000	47051.087566	282589.114509	776942	2007279
Flag Min	1549387.580000	47240.827047	283728.690974	1047358	2234757
ROM Alarm Search	219646.390000	86.402139	518.931768	217890	220951
ROM Match	219657.230000	51.808620	311.162884	218838	220485
Read	109383.320000	18.096356	108.686825	109157	109711
Write	890.550000	0.086548	0.519810	889	891
ns=20					
Flag Max	2009254.370000	59649.611398	358255.924314	1185501	3083556
Flag Min	2012893.160000	58549.964638	351651.439267	989947	2863701
ROM Alarm Search	219418.220000	106.950453	642.345066	217863	220938
ROM Match	219398.630000	46.054170	276.601623	218793	220122
Read	109227.140000	20.640005	123.963991	108982	109522
Write	890.600000	0.081979	0.492366	890	891
ns=25					
Flag Max	2521380.400000	63352.555687	380495.829954	1775328	3507884
Flag Min	2510526.110000	63114.049052	379063.357670	1583172	3290933
ROM Alarm Search	219246.900000	92.229635	553.931741	217847	220424
ROM Match	219209.140000	48.894632	293.661456	218420	219896
Read	109142.770000	18.558406	111.461898	108822	109405
Write	890.590000	0.082303	0.494311	890	891
ns=30					
Flag Max	3041388.650000	69920.271168	419941.568577	1819686	4267795
Flag Min	2992833.630000	69241.216691	415863.163311	1780453	4202045
ROM Alarm Search	219121.770000	109.348570	656.748170	217511	221328
ROM Match	219088.830000	56.342782	338.395085	218034	219854
Read	109080.720000	20.276948	121.783471	108696	109335
Write	890.510000	0.083653	0.502418	890	891

case, the state graph consists of 13 nodes and 17 edges (see Table 2). Figure 6 shows the detailed behavior obtained for the 1-sensor system via its state space. Rectangles with round corners are the states, which are labeled with three numbers (state number and below, the number of predecessors and successors). Edges are labeled as follows: edge number, state transition ($i \to j$), TCPN page, fired transition and binding used.

Transition *reset* can be fired from nodes 1, 5, 6 and 9, which correspond to the initial state (node 1) and the states after *ROM_Search*, *ROM_Match* and either a read or write operation. A cyclic behavior can be observed from nodes 5 and 9. Nodes 6, 8, 9, 11, 12 and 13 are actually home markings, which indicates that it is always possible to return to them. By following the paths in the figure, we can verify the operation of the 1-wire protocol step by step. After performing a reset, from nodes 2, 7 and 11 the *wait_slaves* transition is always fired, followed by the firing of the transition *end_presence*, from nodes 4 and 13. From node 4 we can execute either a *ROM_Alarm_Search* or a *ROM_Match* command. In the first case node 5 is reached, from which a new *reset* must be performed. When a *ROM_Match* command is issued (node 6), one of the three following commands is performed: *read_memory*, *write_memory* or *reset*. Notice

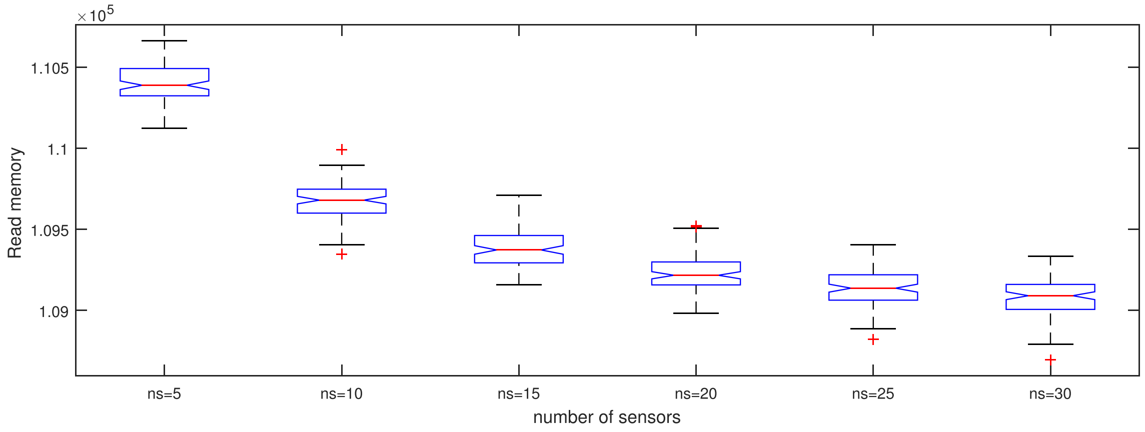

FIGURE 10. ***Read*** **monitor simulation results (100 replications).**

```
Linear model Poly1:
     f(x) = p1*x + p2
Coefficients (with 95% confidence bounds):
    p1 = 1.011e+05  (9.859e+04, 1.036e+05)
    p2 = -5357  (-5.368e+04, 4.297e+04)

Goodness of fit:
  SSE: 1.398e+09
  R-square: 0.9997
  Adjusted R-square: 0.9996
  RMSE: 1.87e+04
```

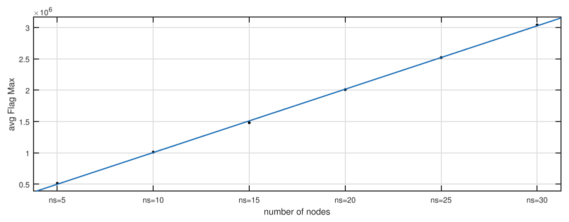

FIGURE 11. **Lineal polynomial fit considering Flag Max vs number of nodes.**

```
General model Power1:
     f(x) = a*x^b
Coefficients (with 95% confidence bounds):
       a =    4.489e+05  (4.471e+05, 4.507e+05)
       b =       -1.008  (-1.01, -1.006)

Goodness of fit:
  SSE: 6408
  R-square: 1
  Adjusted R-square: 1
  RMSE: 40.03
```

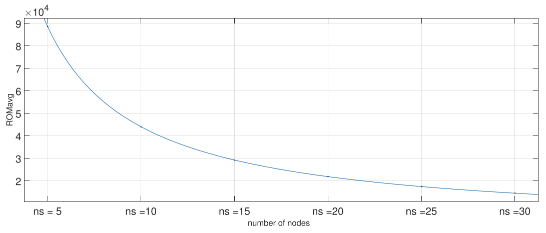

FIGURE 12. **Power Fit considering number of ROMs (ROM Alarm Search plus ROM Match) per sensor vs number of nodes.**

that node 13 corresponds to the same situation as node 4, from which either a *ROM_Alarm_Search* or a *ROM_Match* command can be performed.

From the report in Table 2 we can see that there are no dead markings (deadlocks) and all transitions can eventually be fired (no dead transitions). Actually, all transitions are live, which means that they are eventually fireable from every reachable state. Table 3 shows these same results for five sensors, so the number of sensors does not affect these properties, as expected.

VI. PERFORMANCE EVALUATION

Another important advantage of using CPN Tools is that we can obtain relevant performance measures through simulation experiments, using the monitor features of CPN Tools. In this case, monitors are used to observe, inspect, or control simulations. In particular, we use a breakpoint monitor, which is used to stop a simulation, and data collector monitors, which are used to extract numerical data from a TCPN. The numerical data are then used to calculate statistics.

TABLE 5. **Scenario for the simulations.**

```
val  ti=(20.0,15.0,25.0,0);
  (* Temperature initial marking: mean, minimum
     threshold, maximum threshold *)
val mtn=0.0;
  (* Mean of normal distribution to modify
     temperature *)
val sdtn=3.0;
  (* Standard deviation of normal distribution to
     modify temperature*)
val man=0.0;
  (* Mean of normal distribution to modify alarm
     thresholds *)
val sdam=2.0;
  (* Standard deviation of normal distribution to
     modify alarm thresholds *)
```

The performance evaluation of the protocol is carried out in a generic scenario, since we use probability distributions to generate both the temperature sensor values and threshold values. Our study simulates 15 minutes of operation of the protocol for a system consisting of 5, 10, 15, 20, 25 and 30 sensors. The only change required in the TCPN to set a new

TABLE 6. QNAP2 vs. CPN performance report for 100 replications.

	CPN	QNAP2
ns=5		
Flag Max	514480.35	553137.15
Flag Min	502598.60	552504.11
ROM Alarm Search	221749.15	221561.60
ROM Match	221647.82	221719.57
Read	110401.86	110449.90
Write	890.73	890.73
ns=10		
Flag Max	1012930.63	1099845.62
Flag Min	1010713.07	1103097.64
ROM Alarm Search	220108.06	220196.26
ROM Match	220199.58	220278.32
Read	109678.50	109683.16
Write	890.63	890.67
ns=15		
Flag Max	1481194.47	1648820.22
Flag Min	1549387.58	1643596.75
ROM Alarm Search	219646.39	219619.23
ROM Match	219657.23	219689.43
Read	109383.32	109420.99
Write	890.55	890.58
ns=20		
Flag Max	2009254.37	2193214.36
Flag Min	2012893.16	2192340.08
ROM Alarm Search	219418.22	219430.75
ROM Match	219398.63	219410.79
Read	109227.14	109263.23
Write	890.60	890.62
ns=25		
Flag Max	2521380.40	2734271.51
Flag Min	2510526.11	2735866.16
ROM Alarm Search	219246.90	219132.26
ROM Match	219209.14	219268.90
Read	109142.77	109190.00
Write	890.59	890.64
ns=30		
Flag Max	3041388.65	3289114.57
Flag Min	2992833.63	3285503.94
ROM Alarm Search	219221.77	219129.30
ROM Match	219088.83	219044.38
Read	109080.72	109126.19
Write	890.51	890.54

number of sensors is to assign a new value to the constant *ns* in the TCPN *Declaration* section.

This is an important advantage of the TCPN model, as it is easily configured by just changing a constant. We can easily conduct simulations with any number of sensors and analyze the protocol behavior as the number of sensors increases. In particular, we can check whether the number of sensors affects the protocol performance, in terms of the number of operations it can support on the sensors.

The simulations were run on a computer with an Intel Core i7-6700 3.4GHz processor and 16 GB DDR4 SDRAM, under Windows 10 Enterprise Anniversary Edition x64. The specific scenario that we have considered in terms of initial temperature values and their dynamic updates is shown in Table 5.

Table 4 shows the following relevant information obtained from the simulations:

- *ROM alarm Search* (Figure 7),
- *ROM Match* (Figure 8),
- *Alarms* (Min or Max) (Figure 9),
- *Read memory* commands (Figure 10).

All this information was gathered from data collector monitors by considering the case of count transition occurrence monitors. Table 4 shows the numeric results obtained from CPN Tools, giving the average of each measure of interest together with its confidence intervals.

Figures 7 to 10 depict the information gathered from 100 simulation replications for the ROM Match, ROM alarm Search, Flag Alarms and Read monitors, respectively. These figures show a boxplot of each result, with the median of the number of occurrences and its corresponding first and third quartile, while the whiskers determine minimum and maximum values (without considering the outliers).[4]

Table 4 shows that the number of executed *ROM Alarm Search* and *ROM Match* commands remains practically unchanged for all system configurations. It is obvious that the number of these commands executed is uniformly distributed among all the sensors. An interesting result regarding scalability is that as the number of sensors increases, the number of *ROM* commands tends to stabilize (see Figures 7 and 8). That is to say, adding more sensors does not increase the number of *ROM* commands performed by the Master for the same simulation time interval (15 minutes). The same occurs for the *Read* command (Figure 10). However, in the case of *Flag Alarms*, the increase in the number of sensors results in a higher number of *Flag Alarms* (see Figure 11). This increase, however, does not have a negative impact on the number of ROM commands or read/write commands.

Figure 12 shows the expected number of ROMs commands, that is ROM Alarm Search plus ROM Match, executed by the sensors, which is inversely proportional to the number of sensors (*ns*).

Finally, since the execution of the *Write* command is rather unusual, as according to the considered scenario there is a *Write* command every 1*s*, the number of *Write* commands remains at 890-891 in practically all scenarios.

A. RESULTS IN QNAP

In order to verify the performance results reported by the TCPN model, we developed a simulator using the QNAP2 software [21], which is a discrete-event simulation

[4]Points are drawn as outliers if they are larger than q3 + w(q3 − q1) or smaller than q1 − w(q3 − q1), where q1 and q3 are the 25th and 75th percentiles, respectively. The default of w=1.5 corresponds to approximately a 99.3 coverage if the data are normally distributed.

tool developed in the 80's by a team of researchers at INRIA Labs (France).

Table 6 reports the numerical results obtained using QNAP2 and the ones reported by CPN Tools. As can be seen from the table, the results closely match each other. This confirms that our TCPN model operates properly. From this comparative analysis, we can also highlight the great benefits of CPNs as a modeling methodology. While event-based simulation tools allow us to evaluate performance metrics, e.g., the number of operations performed per unit of time, they do not provide us with the means to describe the concurrent behavior of distributed systems, such as, liveness properties or deadlocks, among others. Petri nets provide a simple graphical format and operational semantics that enable the modeling of the static and dynamic aspects of concurrent systems.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a timed colored Petri net model for the 1-wire protocol. CPN Tools have been used in order to analyze the TCPN model and obtain both qualitative and quantitative results for the 1-wire protocol behavior. The results obtained have also been validated by using a discrete-event simulator (QNAP2).

The proposed TCPN model has been designed to easily expand the number and characteristics of the sensor platform. We can therefore analyze systems with different dimensions, number of sensors, and application domains. Regarding protocol scalability, we have seen that the number of both *ROM_Match* and *ROM_Alarm_Search* commands tends to stabilize as the number of sensors increases, which is due to the initial wait of between 240 μs and 640 μs, which tends towards the upper values of this interval as more sensors are added. However, the number of ROM commands that can be performed on each sensor decreases in inverse proportion to the number of sensors, because the line does not allow more operations, and new operations must wait for the completion of the previous ones.

As future work, we intend to extend the number of ROM commands supported by the model, by including the *Read ROM*, *Skip ROM* and *ROM search* commands. We also plan to apply this generic model to certain specific case studies, such as defining different temperature thresholds or considering various metrics, e.g., temperature and humidity sensors to control a plantation system. Real measured values could then be used to feed the TCPN by using the Comms/CPN features of CPN Tools, and actions could be taken in response to the alarms produced by introducing the corresponding transitions, which could be connected with Java programs performing these actions.

APPENDIX

Table 7 shows the colset declaration section of the TCPN, and Table 8 contains the function declaration section.

The colset declarations (Table 7) are the following:

- B: a bit value (0 or 1).

TABLE 7. TCPN colsets.

```
colset B = int with 0..1;
colset D= int with 0..2;
colset C = int with 0..3;
colset S=product C*INT timed;
colset MS=product REAL*REAL*REAL*D ;
colset LMS=list MS ;
colset INTt=INT timed;
colset M=product INT*INT*B*MS;
colset LM=list M ;
colset UNITt=UNIT timed;
colset REAL3=product REAL*REAL*REAL*D;
colset INTR=product INT*REAL timed
```

- D: an integer between 0 and 2 used in the sensor memory to control a flag value, which is 0 when an alarm is not triggered, 1 when a minimum temperature alarm is triggered and 2 when a maximum temperature alarm is triggered.
- C: an integer between 0 and 3 used in the slaves to annotate the slave states. The possible slave state values are:
  1) 0: initial slave state, no action has been performed.
  2) 1: a reset signal has been received from the master.
  3) 2: the slave has sent the presence pulse to the master.
  4) 3: the slave can perform a memory command.
- S: a timed product of a C and an integer, which models the complete slave state. Their contents are the following:
  1) C: this element allows us to model the slave state, described above.
  2) INT: the slave identifier.
- MS: a timed product of three reals and a D, which defines the sensor memory information. It consists of the measured temperature, the value of minimum and maximum alarms and the flag value for each sensor.
- LMS: a list of the corresponding MS for all the connected sensors.
- INTt: a timed integer.
- M: a timed product of two integers, a B and an MS. It captures the master state, where the first integer indicates whether the presence signals sent by the slaves have been received by the master; the second integer is the slave identifier, and B allows us to capture whether the *ROM command* has been sent or not, and finally MS represents the sensors memory information for each sensor.
- LM: a list of M, which saves the M information for all the connected sensors.
- UNITt: a timed UNIT.
- REAL3: a product of three reals and a D, which models the sensor memory information as explained for MS, but without considering the time for each component.
- INTR: a timed product of an integer and a real, which is used in the *ROM Alarm Search command* in order to log information about the triggered alarms. Specifically,

TABLE 8. TCPN function declaration section.

```
fun testMax(R:REAL)= if R>28.0 then 28.0
                  else if R<22.0 then 22.0 else R;

fun testMin(R:REAL)=if R<12.0 then 12.0
                  else if R>18.0 then 18.0 else R;

fun nt(t3:REAL3)=
 (#1 t3, testMin(#2  t3 + normal(man,sdam)),
 testMax(#3 t3 + normal(man,sdam)),#4 t3);

fun newalarms(ns:INT, tp:LM)= if ns=1 then
[(#1 (hd tp),ns,1,nt(#4 (hd tp)))]
 else
  newalarms(ns-1,tl tp)^^
  [(#1 (hd tp),ns,1,nt(#4 (hd tp)))];

fun sstate(ns:INT,v:C)=
     if ns=1 then 1`(v,ns)
     else 1`(v,ns)++sstate(ns-1,v);

fun MtempS(ns:INT)= if ns=1 then 1`ns
           else 1`ns++MtempS(ns-1);

fun MMastern(ns)= if ns=1 then [(0,1,0,ti)]
   else  [(0,ns,0,ti)]^^MMastern(ns-1);

fun MSM(ns)=if ns=1 then [ti] else  MSM(ns-1)^^ [ti];

fun generateids(ns:INT)= if ns=1 then 1`1 else 1`ns++generateids(ns-1);

fun fpresence(me:LM,id:INT)= if length me=0 then nil
         else if id=(#2 (hd me)) then [(1, #2 (hd me),#3 (hd me),#4 (hd me))]^^tl me
         else  [hd me]^^fpresence(tl me,id);

fun fROM(me:LM,id:INT)= if length me=0 then nil
         else if id=(#2 (hd me)) then [(#1 (hd me), #2 (hd me),1,#4 (hd me))]^^tl me
         else  [hd me]^^fROM(tl me,id);

fun fRead(lms:LMS,me:LM,id:INT)= if length me=0 then nil
         else if  id=(#2 (hd me)) then [(#1 (hd me), #2 (hd me),#3 (hd me),List.nth(lms,(id-1)))]^^tl me
         else  [hd me]^^fRead(lms,tl me,id);

fun fWrite(lms:LMS,me:LM,id:INT)= if length me=0 then nil
         else if  id=(#2 (hd me)) then [(#1 (hd me), #2 (hd me),#3 (hd me),List.nth(lms,(id-1)))]^^tl me
         else  [hd me]^^fWrite(lms,tl me,id);

fun temper(temp)=temp+normal(mtn,sdtn);

fun setTemper(lms:LMS,id:INT,nb:INT)= if length lms=0 then nil
         else if (nb=id) then
                [(temper(#1 (hd lms)), #2 (hd lms), #3 (hd lms), #4 (hd lms))]^^setTemper(tl lms,id,nb+1)
              else  [(#1 (hd lms), #2 (hd lms), #3 (hd lms), #4 (hd lms))]^^setTemper(tl lms,id,nb+1);

fun setFlag(lms:LMS)=
    case length lms of
    0 => nil
    |_=> if (#1 (hd lms)<(#2 (hd lms))) then
           [(#1 (hd lms), #2 (hd lms), #3 (hd lms), ALARM_MIN)]^^setFlag(tl lms)
         else if (#1 (hd lms)>(#3 (hd lms))) then
                   [(#1 (hd lms), #2 (hd lms), #3 (hd lms),ALARM_MAX)]^^setFlag(tl lms)
              else  [(#1 (hd lms), #2 (hd lms), #3 (hd lms), 0)]^^setFlag(tl lms);

fun searchMin(lms:LMS,nb:INT)=
     case length lms of
     0 => empty
     |_=> if (#4 (hd lms))=1 then 1`(nb,#1 (hd lms))++searchMin(tl lms, nb+1)
          else searchMin(tl lms, nb);

fun searchMax(lms:LMS,nb:INT)=
     case length lms of
     0 => empty
     |_=> if (#4 (hd lms))=2 then 1`(nb,#1 (hd lms))++searchMax(tl lms, nb+1)
          else searchMax(tl lms, nb);
```

the slave identifier and the temperature that produced the alarm.

The TCPN function declarations are the following (Table 8):

- testMax(), testMin(): They have a real parameter, and are used in the *nt* function in order to set a maximum and minimum values for the generated new temperature values obtained in *nt*. They are used in the *nt* function.
- nt(): This function is used in the *newalarms* function to generate new minimum and maximum values for a sensor alarm from the sensor information (*REAL3*). This function has a *REAL3* variable type (sensor information) as parameter. It is used in the *newalarms* function.
- newalarms(): It is a recursive function that has two parameters: an integer and an *LM* variable. Its purpose is to go through the list elements and to change the maximum and minimum alarm values for all the sensors in the list LM, and for this purpose it calls the *nt* function.
- sstate(): It is a recursive function that has two parameters: an integer and a *C* variable. Its purpose is to change all sensor states.
- MtempS(): It sets the initial marking for the *Control Memory* place, which is used to set the times at which the temperature values of sensors are updated.
- MMaster(ns): It sets the initial marking for the master.
- MSM(): It sets the initial marking for the sensor information (*Sensor_Memories* place).
- generateids(): It generates the sensor identifiers. It is used to wait until all sensors have received the reset signal (*End Wait* place).
- fpresence(): This function has two parameters: an *LM* variable and an integer (sensor identifier). It is a recursive function that goes through the *LM* list, looking for the sensor with the given identifier and sets to 1 the second element of *M* for this sensor, to indicate that the corresponding sensor has sent the presence signal.
- fROM(): This function has two parameters: an *LM* variable and an integer (sensor identifier). It is a recursive function that goes through the *LM* list looking for the sensor with the given identifier and sets to 1 the fourth element (B type) of *M* for this sensor, to indicate that the *ROM Match* has been performed for this sensor.
- fRead(): This function has three parameters: an *LMS* variable, an *LM* variable, and an integer (sensor identifier). It is a recursive function that goes through the *LM* list, looking for the sensor with the given identifier, and modifies the fifth element (MS type) of *M* for this sensor, with the sensor information for this sensor obtained from the variable *LMS*, which contains the updated sensor information saved in the Sensor Memories place.
- fWrite(): This function is similar to the *fRead* function, but in this case, the fifth element (MS type) of the corresponding sensor is updated with the sensor information saved in the master.
- temper(): This function has a real parameter and is used in the *setTemper* function. Its purpose is to calculate a new temperature value by using a normal distribution function and the real parameter. It is used in the *setTemper* function.
- setTemper(): This function uses three parameters: an LMS variable and two integers (sensor identifier and integers used for recursive function calling). It goes through the list elements of LMS looking for the sensor with the given identifier, and it changes the temperature value for this sensor.
- setFlag(): This function has one parameter of *LMS* type. It goes through the LMS list and checks for each element in the list, whether the temperature value is less than the minimum alarm or greater than the maximum alarm in order to modify the flag value.
- searchMin(), searchMax(): These functions have two parameters: an LMS variable and an integer (sensor identifier). They allow us to log information in the form of (identifier, temperature) at the places *FlagMin* and *FlagMax*, when a minimum or maximum alarm are triggered, respectively.

REFERENCES

[1] M. A. Azgomi and A. Khalili, "Performance evaluation of sensor medium access control protocol using coloured Petri nets," *Electron. Notes Theor. Comput. Sci.*, vol. 242, no. 2, pp. 31–42, 2009.

[2] B. Huang, J. Lei, and Y. Bo, "The reading data error analysis of 1-wire bus digital temperature sensor DS18B20," in *Proc. Int. Conf. Modeling, Identificat. Control*, Jun. 2012, pp. 433–436.

[3] J. Ben-Othman, S. Diagne, L. Mokdad, and B. Yahya, "Performance evaluation of a hybrid MAC protocol for wireless sensor networks," in *Proc. 13th Int. Symp. Modeling Anal. Simulation Wireless Mobile Syst. (MSWiM)*, Bodrum, Turkey, Oct. 2010, pp. 327–334.

[4] J. Billington and S. Vanit-Anunchai, "Coloured Petri net modelling of an evolving Internet standard: The datagram congestion control protocol," *Fundam. Inform.*, vol. 88, no. 3, pp. 357–385, 2008.

[5] J. Billington and C. Yuan, "On modelling and analysing the dynamic MANET on-demand (DYMO) routing protocol," in *Transactions on Petri Nets and Other Models of Concurrency III*. Berlin, Germany: Springer, 2009, pp. 98–126.

[6] M.-T. Chew, T.-H. Tham, and Y.-C. Kuang, "Electrical power monitoring system using thermochron sensor and 1-wire communication protocol," in *Proc. 4th IEEE Int. Symp. Electron. Design, Test Appl. (DELTA)*, Jan. 2008, pp. 549–554.

[7] J. Dudak and P. Cicak, "CPN model of the MODBUS protocol," in *Proc. 13th Mechatronika*, Jun. 2010, pp. 43–45.

[8] R. O. Gosheblagh and K. Mohammadi, "Designing and implementing a reliable thermal monitoring system based on the 1-wire protocol on FPGA for a LEO satellite," *Turkish J. Electr. Eng. Comput. Sci.*, vol. 23, no. 1, pp. 171–186, 2015.

[9] A. Heindl and R. German, "Performance evaluation of IEEE 802.11 wireless LANs with stochastic Petri nets," *Perform. Eval.*, vol. 44, nos. 1–4, pp. 139–164, 2001.

[10] X. Hu and L. Jiao, "Efficient modeling and performance analysis for IEEE 802.15.4 with coloured Petri nets," in *Proc. 25th IEEE/ACM Int. Symp. Quality Service (IWQoS)*, Vilanova i la Geltrú, Spain, Jun. 2017, pp. 1–6.

[11] Maxim Integrated. (2002). *White Paper 5: Using 1-Wire APIS for Data Sheet Commands*. [Online]. Available: https://www.maximintegrated.com/en/appnotes/index.mvp/id/1100

[12] Maxim Integrated. (2017). *DS1822-PAR ECONO Parasite-Power Digital Thermometer*. [Online]. Available: https://datasheets.maximintegrated.com/en/ds/DS1822-PAR.pdf

[13] K. Jensen and L. M. Kristensen, *Coloured Petri Nets—Modelling and Validation of Concurrent Systems*. Berlin, Germany: Springer, 2009.

[14] L. M. Kristensen and K. I. F. Simonsen, "Applications of coloured Petri nets for functional validation of protocol designs," in *Transactions on Petri Nets and Other Models of Concurrency VII*, K. Jensen, W. M. P. van der Aalst, G. Balbo, M. Koutny, and K. Wolf, Eds. Berlin, Germany: Springer, 2013, pp. 56–115.

[15] H. Macià, M. C. Ruiz, J. A. Mateo, and J. L. Calleja, "Petri nets-based model for the analysis of NORIA protocol," *Concurrency Comput., Pract. Exper.*, vol. 27, no. 17, pp. 4704–4715, 2015.

[16] H. Macià, V. Valero, G. Díaz, J. Boubeta-Puig, and G. Ortiz, "Complex event processing modeling by prioritized colored Petri nets," *IEEE Access*, vol. 4, pp. 7425–7439, 2016.

[17] Maxim Integrated. (Mar. 2018). *Guidelines for Reliable Long Line 1-Wire Networks*. [Online]. Available: https://www.maximintegrated.com/en/appnotes/index.mvp/id/148

[18] Maxim Integrated. (Mar. 2018). *Overview of 1-Wire Technology and its Use*. [Online]. Available: https://www.maximintegrated.com/en/appnotes/index.mvp/id/1796

[19] S. Muzaffar, N. Saeed, and I. M. Elfadel, "Automatic protocol configuration in single-channel low-power dynamic signaling for iot devices," in *Proc. IFIP/IEEE Int. Conf. Very Large Scale Integration (VLSI-SoC)*, Tallinn, Estonia, Sep. 2016, pp. 1–6.

[20] M. D. R. Perera, R. G. N. Meegama, and M. K. Jayananda, "FPGA based single chip solution with 1-wire protocol for the design of smart sensor nodes," *J. Sensors*, vol. 2014, pp. 125874:1–125874:11, Nov. 2014.

[21] D. Potier, "New users' introduction to QNAP2," INRIA, Rocquencourt, France, Tech. Rep. 40, 1984.

[22] Microchip Technology Inc. (2008). *Sashavalli Maniyar. 1-Wire Communication With PIC Microcontroller*. [Online]. Available: http://ww1.microchip.com/downloads/en/AppNotes/01199a.pdf

[23] J. T. Weng and M. A. Ameedeen, "A survey of Petri net tools," in *Advanced Computer and Communication Engineering Technology*, H. A. Sulaiman, M. A. Othman, M. F. I. Othman, Y. A. Rahim, and N. C. Pee, Eds. Springer, 2015, pp. 537–551.

[24] (Mar. 2018). *CPN Tools*. [Online]. Available: http://cpntools.org/

[25] S. Vanit-Anunchai, J. Billington, and G. E. Gallasch, "Analysis of the datagram congestion control protocol's connection management procedures using the sweep-line method," *Int. J. Softw. Tools Technol. Transf.*, vol. 10, no. 1, pp. 29–56, 2008.

[26] S. Zairi, A. Mezni, and B. Zouari, "Formal approach for modeling, verification and performance analysis of wireless sensors network," in *Proc. 13th Int. Conf. Wired/Wireless Internet Commun. (WWIC)*, Malaga, Spain, May 2015, pp. 381–395.

[27] H. Zhang, H. Zhang, M. Gu, and J. Sun, "Modeling a heterogeneous embedded system in coloured Petri nets," *J. Appl. Math.*, vol. 2014, 2014, Art. no. 943094, doi: https://doi.org/10.1155/2014/943094

[28] F. Zhang, Y. Xu, and J. Chou, "A novel Petri nets-based modeling method for the interaction between the sensor and the geographic environment in emerging sensor networks," *Sensors*, vol. 16, no. 10, p. 1571, 2016.

MARÍA EMILIA CAMBRONERO received the Ph.D. degree from the University of Castilla-La Mancha, Spain, in 2007. She was an Assistant Professor with the University of Castilla-La Mancha for several years, where she is currently an Associate Professor of computer science with the Computer Science School of Albacete. Her research interests are aimed to make software more reliable, more secure, and easier to design. Her primary technical interests include software engineering and related areas, including contract specification, program monitoring, testing, and verification. Her research combines strong theoretical foundations with realistic experimentation in the area of web services and cloud computing. She received the Tenure Distinction in 2012.

HERMENEGILDA MACIÀ received the degree in mathematics from the University of Valencia and the Ph.D. degree in computer science from the University of Castilla-La Mancha, Spain, in 2003. She is currently an Associate Professor with the Department of Mathematics, Computer Science School of Albacete, University of Castilla-La Mancha. She has published research articles in reputed journals of mathematics and computer science. Her main research interests include the theoretical study and applications of formal methods, such as process algebras and petri nets considering timed, probabilistic, and stochastic extensions.

VALENTÍN VALERO received the degree in mathematics from the Complutense University of Madrid in 1987 and the Ph.D. degree in mathematics from the Department of Computer Science, Complutense University of Madrid, in 1993. Since 1987, he has been a member of the Computer Science Department, University of Castilla-La Mancha, Spain, where he is a Full Professor of distributed systems and operating systems with the Computer Science School of Albacete. His current research interests include concurrency, specifically in formal models for analysis and design of concurrent systems and real-time systems.

LUIS OROZCO-BARBOSA (M'81) received the B.Sc. degree in electrical and computer engineering from Universidad Autónoma Metropolitana, Mexico, in 1979, the Diplomed'études approfondies in computer science from the École Nationale Supérieure d'Informatique et de Mathématiques Appliquées, France, in 1984, and the Doctorat d'Université in computer science from Université Pierre et Marie Curie, France, in 1987. From 1991 to 2002, he was a Faculty Member with the School of Information Technology and Engineering, University of Ottawa, Canada. In 2002, he joined the Department of Computer Engineering, Universidad de Castilla La Mancha, Spain. He has also been appointed as the Director of the Albacete Research Institute of Informatics, a Regional Centre of Excellence. He has conducted numerous research projects with the private sector and served as a Technical Advisor for the Canadian International Development Agency and the Spanish International Cooperation Council. His current research interests include Internet protocols, wireless sensor communications, and IoT technologies.