IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 26 July 2022, accepted 25 August 2022, date of publication 5 September 2022, date of current version 14 September 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3204043

RESEARCH ARTICLE

Modeling and Verification of Symbolic Distributed Applications Through an Intelligent Monitoring Agent

ANDREA AUGELLO[1], SALVATORE GAGLIO[1,2], (Life Member, IEEE), GIUSEPPE LO RE[1], (Senior Member, IEEE), AND DANIELE PERI[1]

[1]Department of Engineering, University of Palermo, 90128 Palermo, Italy
[2]Institute for High Performance Computing and Networking (ICAR), National Research Council (CNR), 90146 Palermo, Italy

Corresponding author: Daniele Peri (daniele.peri@unipa.it)

This work was supported by the Project "S6 Project (A Smart, Social and SDN-based Surveillance System for Smart-Cities)," (PO FESR 2014/2020) Regione Siciliana, Italy.

ABSTRACT Wireless Sensor Networks (WSNs) represent a key component in emerging distributed computing paradigms such as IoT, Ambient Intelligence, and Smart Cities. In these contexts, the difficulty of testing, verifying, and monitoring applications in their intended scenarios ranges from challenging to impractical. Current simulators can only be used to investigate correctness at source code level and with limited accuracy. This paper proposes a system and a methodology to model and verify symbolic distributed applications running on WSNs. The approach allows to complement the distributed application code at a high level of abstraction in order to test and reprogram it, directly, on deployed network devices. The proposed intelligent architecture enables the execution of distributed applications and the verification of the supplied correctness conditions. This paper shows the feasibility of the proposed approach and its effectiveness even when networks include resource-constrained nodes with some sample applications and quantitative experiments measuring the overhead introduced by the monitoring operations.

INDEX TERMS Distributed applications, embedded systems, fault detection, Internet of Things, knowledge based systems, software monitoring, wireless sensor networks.

I. INTRODUCTION

Emerging distributed computing paradigms, such as the Internet of Things (IoT), Ambient Intelligence, and Smart Cities, envision complex distributed applications running on heterogeneous networks of devices ranging from resource-constrained to specialized high-performance ones. Wireless Sensor Networks (WSNs), which can be easily seen as foundational for these paradigms, are usually composed of nodes of the former type, providing a baseline for evaluating methodologies to develop distributed applications in the aforementioned contexts.

Traditionally, WSNs have been successfully employed in many applications such as smart farming, transport control, and industrial process management [1], to name a few. The potential of their integration in the IoT paradigm has been shown in the context of smart home [2] and healthcare monitoring [3] to improve quality of life and safety [4]. For instance, applications for localization and tracking of objects and people [5], e.g. through range-free connectivity information, multilateration, and angulation [6] are particularly useful for security and safety purposes. Also, Smart Cities can benefit from pervasive data collection provided by WSNs for vehicular traffic and congestion management through traffic lights control [7].

Setting up a distributed application running on a large number of devices, which store, process, and exchange data, imposes an accurate selection among different networks, applications, and data protocols. However, evaluating the performance and comparing distributed applications in different real scenarios is far from being easy. Physical-world dynamics, device mobility, and resource constraints

The associate editor coordinating the review of this manuscript and approving it for publication was Shaohua Wan.

often make the repeatability of experiments in real contexts impractical [8].

Simulated environments ensure test reproducibility in a precise way, but fail in capturing and integrating physical world phenomena [9]. As simulation-based solutions depend on the evaluation environment, models and simulation parameters, misleading results could be observed for the same application when the simulation tool changes [10].

For accurate verification results, execution models of single nodes also need to be taken into account, especially when resource-constrained devices are considered. Namely, simulations do not include processing delays, for example for packet processing, while substantial differences in the execution time between real and emulated resource-constrained nodes are quite common [11].

Testbeds enable prototyping and verifying distributed applications in large and realistic environments [12], [13], [14], [15]. They are sometimes deployed in smart cities [16], however they are usually limited to restricted and controlled areas [17]. Nevertheless, management and scheduling functionalities, which are usually implemented for testbeds, are strictly bound both to the specific deployment and to the sensor node operating system. This makes reusability particularly unfeasible [18].

Simulators have been diffusely used to evaluate the performance of distributed applications for WSNs [19], [20], [21]. However, the main limit of these tools is that the application under test must be adapted to run in the simulation environment on hardware which is generally very different from the targeted one. The compliance of the application to the specifications can thus only be tested before deployment.

Letting some real nodes be integrated in simulations is the strategy adopted in hybrid verification. However, the resulting mixture of real and virtual nodes makes timing problems more and more evident, especially in large networks [22].

All of these solutions put in place post processing methods to perform evaluation at the end of the implementation work, before the network is deployed in a real-world environment. However, the capability of verifying applications at each stage of design, deployment, and implementation is desirable [23].

In particular, after a network has been deployed for an extended period of time, its characteristics may be substantially different from the testing condition during implementation and initial evaluation. Typical variations in the network behavior are due to nodes running out of power [24], suffering failures in their communication modules, or otherwise malfunctioning [25].

Moreover, WSNs are often deployed in inaccessible areas, or may comprise thousands of nodes. In both cases, manually checking that each node behaves properly may simply be unfeasible. Thus, post-deployment [26] monitoring systems are valuable tools to diagnose malfunctions without stopping the system and to acquire information about the WSN operation [27].

In this work, we present a WSN monitoring platform for the debugging of symbolic distributed applications. The main contributions and novelties of this work are:

- the proposed system does not require any extra preinstalled debugging code or hardware on the deployed nodes, reducing the burden on resource-constrained devices, and potentially prolonging the WSN lifespan and leaving more resources available for ordinary operations;
- the interactive approach enabled by symbolic programming allows multiple verification modes that can be added to the system long after the network has been deployed with greater flexibility than currently existing solutions;
- the symbolic computation model permits verification operations on heterogeneous devices unlike the platform-specific tools commonly used;
- knowledge on the network and modeling of the distributed application are used to automatically verify whether the application was executed correctly, without the need for human intervention in monitoring network logs.

The software platform used in this work is DC4CD [28]. This platform was proposed to enable the development of distributed applications on resource-constrained WSN nodes through executable high-level code exchange [28].

In this work, symbolic computation plays a key role in the evaluation of distributed applications during their execution on deployed devices. To this end a rule-based modeling and verification system is introduced.

The proposed system relies on:

1) a knowledge base that includes applications and network specifications and ties application operations to the corresponding verification code;
2) an intelligent agent that uses inference rules to concatenate snippets of symbolic code in the knowledge base to produce verification messages;
3) a communication module that sends application and verification code to deployed nodes;
4) a symbolic verifier that checks that results satisfy expectations and collects necessary metrics.

Development and verification of an application are both performed in terms of executable symbols that are exchanged among entities. Symbolic test programs are executed on the deployed devices as soon as they are received.

The rest of the article is organized as follows. Section II goes over some related works. Section III details the computational paradigm, the architecture of the modeling and verification system, and describes the system operation. Section IV presents case studies concerning various applications. Section V presents experimental results. Finally, Section VI reports our conclusions and discusses future research directions.

II. RELATED WORKS

In this section we discuss related research on WSN monitoring and debugging platforms, and the use of symbolic programming to overcome some of their limitations.

A. MONITORING PLATFORMS

Some WSN monitoring platforms have been presented in the literature. Sensor Network Managing System (SMNS) [29] is one of the first WSN monitoring platforms. It was developed for the TinyOS environment with the design goals of minimal memory footprint and reduced network traffic overhead. Its networking architecture supports collection of health data from the network and dissemination of management commands and queries following a predefined schema.

The authors of [30] proposed a passive WSN monitoring tool. A sniffer device captures packets, timestamps them, and forwards them to an analysis software. The developed interface shows the contents of the packets along with metadata and the tool also monitors link quality. However, the analysis is limited to the network characteristics, leaving out the actual distributed application.

Hybrid Monitoring Platform (HMP) [31] is a hybrid (hardware/software, active/passive) WSN monitoring platform. Three main components make up this system: 1) monitor nodes that record events and related metadata from each node; 2) sniffer nodes that cover portions of the WSN and capture transmissions; 3) a monitor server that collects the information obtained. The monitor nodes are connected to the WSN nodes and require special-purpose code for the node to send data through a wired interface. This extra code increases the memory occupation by about 1 KB. The monitor server only displays the acquired trace ordered by timestamp.

B. REMOTE DEBUGGERS

Clairvoyant [32] is a GDB-based source-level remote debugger that works through dynamic binary instrumentation. The use of this tool entails numerous flash rewrites, which may wear out the node storage eventually leading to shortened network lifetime. Moreover, it occupies 32 KB of program memory and 1 KB of data memory, making the debugging of large programs impossible on resource-constrained nodes.

In [33] the remote source-level debugger approach is proposed. Correct application behavior is verified by running a simulation of a single remote node on a host PC. The simulation is updated using sensing data packets retrieved from the remote node and is kept in sync through ''clock'' packets. Behavior of the remote node is compared with the simulated execution of the same code on the host. This approach generates frequent traffic throughout the distributed application execution and can only monitor a single node.

In the Stethoscope [34] debugging system every deployed node has a debug agent that receives, interprets, and executes debugging commands sent by the command generator running on the sink node. Debugging commands work by changing the address of indirect function calls to run some preprogrammed debugging routine before normal actions. Interrupts are hijacked so that they do not interfere with debugging operations. The proposed debugger occupies 10 KB of flash memory compared to the 33 KB commonly occupied by GDB-based debuggers.

The HDF Hybrid Debugging Framework [35] uses external debugging devices (D2-Box) in a one-to-one mapping to the nodes of the WSN. The D2-Boxes act as debug agents and record data from the sensor nodes, and can send control signals to the node. The debug agents have a wired connection to their sensor node and use a separate communication channel from that of the WSN. Debug agents can also reprogram the connected node through JTAG. This approach can minimize intrusion on the network, however the external debugging device imposes an extra burden on the limited WSN node energy and two separate communication channels are needed.

C. CURRENT LIMITATIONS AND THE SYMBOLIC PROGRAMMING APPROACH

Overall, the monitoring platforms proposed so far present some issues that limit their use cases:

- extra debugging hardware on the nodes increases energy consumption and costs;
- the debugging code on the nodes occupies memory and storage that might be needed for the correct functioning of the distributed applications in execution on the nodes;
- the debugging operations are specified at deployment time and cannot be changed later on without reprogramming the nodes, limiting the effectiveness of these platforms when the WSN incurs in unforeseen issues;
- frequent flash write/erase cycles shorten the lifetime of nodes;
- the data obtained from the nodes need to be analyzed and interpreted separately.

The underlying limitation from which most of these issues stem is that the development and deployment of WSNs usually follows the flashing-rebooting-reloading cycle. To overcome these issues, symbolic distributed computation has been proposed as a promising solution that naturally supports interactive programming. This approach makes reprogramming deployed networks feasible [36] and, in turn, also permits tests to take place both during and after development, even on resource-constrained devices [37], [38]. The complexity of embedded system programming is supported by a high level of abstraction provided by existing software platforms [28]. Moreover, accurate monitoring and verification of individual devices is also feasible at runtime [39]. Additionally, adoption of a symbolic development platform in WSNs would augment the scope of activities that can be performed by a mobile agent in maintenance operations. For instance, Unmanned Aerial Vehicle (UAV)-enabled maintenance of WSNs has been proposed to recharge WSN nodes [40]. Besides these activities, by exploiting symbolic distributed computation a UAV could also act as a probing node able to query status information and even reprogram nodes on the fly.

Symbolic programming has been already adopted to undertake interactive experiments and to verify real hardware instruments through the use of an interpreter [41] and for complex activities such as automatic program synthesis [42].

Evaluation tools for symbolic distributed applications in real scenarios using actual target hardware have been still largely underexplored. As a contribution to filling this gap, this work presents an approach supporting verification of symbolic distributed applications during their execution on real hardware.

III. INTELLIGENT MONITORING SYSTEM

In the following, the proposed system for application modeling and verification is described in both functional and structural terms. Preliminary, the symbolic environment, adopted for the network programming, is presented. Finally, some implementation details are discussed.

A. SYMBOLIC EXECUTION PLATFORM

Deployed nodes run the DC4CD symbolic environment, which is based on the Forth stack-oriented language. Programs are composed of a sequence of symbols, either words or numeric constants. A program is executed by an interpreter which evaluates symbols in the order they are provided. Numeric constants are pushed on top the stack, words are used to search for executable code in a data structure called *word dictionary*. The execution of a word returns values by leaving them on top of the stack. For instance, a sensor could have the word `temperature` defined in its dictionary so that its execution would leave the measured temperature value on top of the stack.

Words can also access the values left on the stack and consume them. Thus, through this stack mechanism words can pass each other parameters. For instance, when the Forth interpreter evaluates the code

```
2 3 +
```

2 and 3 are pushed on the stack, then the symbol +, which is a predefined standard Forth word, is executed, popping the two operands and leaving on top of the stack the result of the addition which can then be used for further computation. The evaluation process follows the postfix notation.

The dictionary can be easily expanded with user-defined words. A user-defined word consists of a chain of words already in the dictionary, the evaluation of such a word causes the sequential execution of the words in its definition. *Colon* (:) and *semicolon* (;) are the Forth words to start and end the definition of a new word in terms of a sequence of already-defined executable words. As a simple example, let us define the word `twotimes` to double the value currently on top of the stack:

```
: twotimes     dup + ;
```

Also `dup` is a standard Forth word. Its execution duplicates the top stack item. The stack effects caused by the execution of `4 twotimes` are shown in Fig. 1.

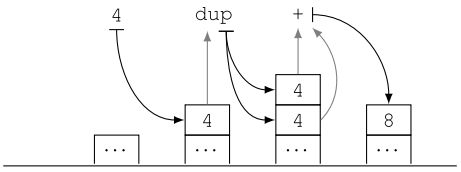


FIGURE 1. Effects of the stack execution of `4 twotimes`. The symbol `4` is recognized as a numeric value and placed on the stack. Then the word `twotimes` is expanded into `dup +`. The value on top of the stack is duplicated by `dup`. The word `+` executes the arithmetic operation and leaves the result on top of the stack.

The dictionary can be be implemented as a linked list of word definitions allocated one after another in memory, the last pointing to the previously defined one. This way the word lookup mechanism can be implemented as a simple backward search from the last to the first definition. Special symbols, called *markers*, can be used to create restore points in the word dictionary. When a marker word is defined, it is placed at the current end of the dictionary as any new user-defined word would be. The execution of a marker removes from the dictionary stack the marker itself and all the words that were defined after, rolling back the dictionary to the state it had right before the definition of the marker. Variables are also stored in the dictionary, their execution leaves their memory address on top of the stack. Values can be read from a memory address through the *fetch* (@) word and written to a specified memory location using the *store* (!) word.

In order to support distributed applications, and to facilitate the exchange of symbolic code, the DC4CD platform natively provides support to distributed computing schemes through a special-purpose construct:

```
tell: <symbolic code to be sent> :tell
```

These two words build IEEE 802.15.4-2003-compliant messages, as required by the node radio (see Section V), containing the symbolic code enclosed between them as payload of datalink level packets. The sequence of words to be sent are encoded as plain ASCII characters.

When the `tell:` word is executed, it consumes the value on top of the stack, and interprets it as the MAC address of the receiver node, placing that value in the destination address field of the packet. Upon receiving a message, the destination node immediately executes the received instruction without any further translation step. The same words might be defined differently depending on the underlying hardware of a node. Each node then executes the words received through messages using the definition in its own dictionary. Exchanging executable symbolic code thus abstracts the characteristics and the representation of target hardware easing interoperability on networks composed of heterogenous devices.

Inside the `tell: :tell` construct, the symbol *tilde* (~) is treated as a symbolic placeholder and replaced with the value currently on the top of the stack. This mechanism permits to include computed values in outbound messages. For example, to tell a remote node with id `7` to measure the

temperature, send the response back (`reply`), and make the requesting node print its value, the requesting node executes:

```
7 tell: temperature reply tell:
                          ~ . :tell :tell
```

where the word *dot* (.) pops the topmost value of the stack and prints it. The effects of the stack execution of this code are shown in Fig. 2. The code is sent as a sequence of words, as simple text. The `tell` construct can send any Forth code: nested `tell`s, definitions of new words, markers, and any arbitrary command. This mechanism allows nodes to exchange data, symbolic rules, simple commands, or complete algorithms.

Through a similar mechanism, nodes can perform multi-hop communication relying on the routing tables of the intermediate nodes using the `forward: :forward` construct.

B. MONITORING MODES

Considering the symbolic computational paradigm presented in the previous subsection, a distributed application is defined as sets of sequence of executable symbols in a concatenative programming model supported by all the nodes of the network.

The aim of our work is to allow an accurate monitoring of distributed application execution in order to discover undesired behaviors and errors through the use of a monitoring agent. The monitoring agent is described in the following sections.

The system can perform verification both during and at the end of execution. The proposed architecture implements four monitoring modes:

- *Targeted*: at the end of execution a single request to retrieve the values of the local variables is transmitted to one of the network nodes that is known to be reliable. Verification consists in checking that the number of received values is correct and that the values of the variables coincide with the ones computed by the monitoring agent with available *a priori* knowledge.
- *Global*: at the end of the execution all the nodes are requested to send the values of some local variables. Then, the monitoring agent waits for the responses. On their arrival, the agent verifies that all the tuples of retrieved values are consistent with each other and with the prior available knowledge.
- *On demand*: each node is queried both during the execution of the application under test, possibly more than once, and then at the end of the execution as in the global monitoring mode. This strategy enables early failure detection by monitoring the evolving state of some or all nodes while they are executing the application.
- *Stepwise*: similarly to the *on demand* strategy, nodes are queried during the execution. In this strategy, no message to start the application execution is sent. Instead, nodes receive executable code from the monitoring agent to execute a single step of the distributed application. The verification proceeds as in the *on demand* case. This strategy overcomes the difficulties of monitoring the state of the network while it is actively changing due to a running application.

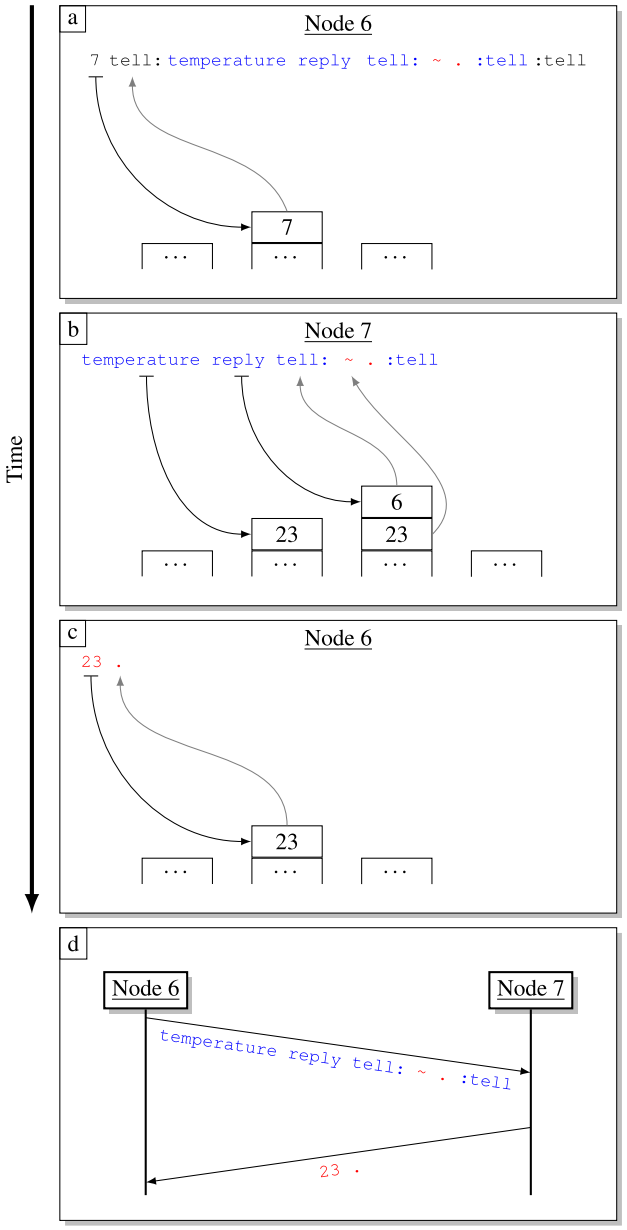


FIGURE 2. Stack execution of the `7 tell: temperature reply tell: ~ . :tell :tell` symbolic code on Node 6: a) the symbol 7 is recognized as a numeric value and placed on the top of the stack, then the outer `tell: ...:tell` construct uses this value as destination node address making Node 6 send the inner code (`temperature reply tell: ~ . :tell`) to Node 7; stack on Node 6 gets back to the initial state; b) Node 7 receives the code and executes it; `temperature` leaves a temperature reading (23) on the stack; `reply` pushes the address of the sender of the received message on the top of the stack; the inner `tell: ...:tell` construct uses this value to send Node 6 a message containing the reading (23), extracted by the tilde (~) placeholder, followed by the word dot (.); c) Node 6 receives `23 .` ; the symbol `23` is again recognized as a numeric value and put on the top of the stack, then the word dot (.) uses this value to produce an output representation of the reading. The exchanged messages are shown in d).

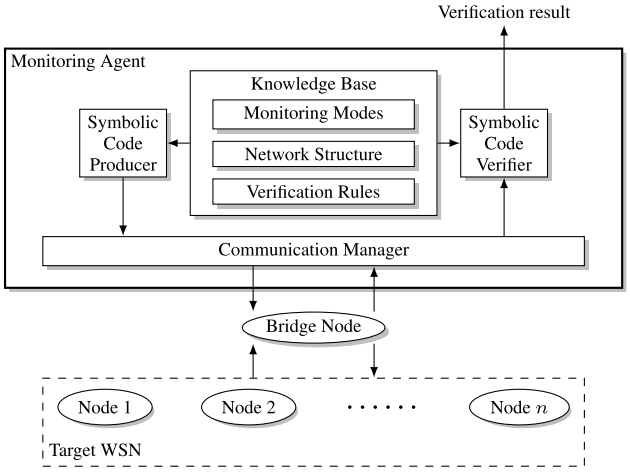


FIGURE 3. Functional blocks composing the verification system and its interaction with the target WSN running the application under test.

Moreover, it enables on-demand verification for applications where the execution order is nondeterministic or nodes send messages too frequently to execute queries without incurring in collisions. This verification scheme can be used to perform record-and-replay debugging [43].

C. SYSTEM ARCHITECTURE

We exploited logic programming in order to implement the monitoring and verification agent.

The modular system architecture is shown in Fig. 3. An intelligent monitoring agent represents the core element of the system, and is in turn composed by a Knowledge Base (KB) and several subagents.

From a structural perspective, the system includes three main components: a Rule System implementing the monitoring agent running on a host computer, a bridge node, and the network of deployed nodes.

The KB maintains both the static knowledge on the domain (network description) and that acquired during the application execution (verification rules and partially inferred results). The distributed application programmers, in the wake of Dijkstra's 1975 ideas [44], together with the code, provide sets of predicates (verification rules) that characterize the intermediate and final states of the execution of the application on each node. The executable application code is stored directly on the nodes as symbolic programs, while the monitoring agent operates on the predicates in the KB.

The monitoring agent is responsible, on the basis of the operating mode set, to choose the appropriate verification rules and to generate the equivalent symbolic code to be executed on the nodes for the purpose of monitoring the operations. The interaction between the verification system and end devices is enabled by the Communication Manager (CM), which acts as a bidirectional interface with already deployed end devices. The CM interacts with the network through a bridge node, which is physically connected to the monitoring agent. Nodes exchange executable symbolic code through the device wireless communication interface. The bridge node has the same specifications of the other nodes in the network, and communicates with the CM through a wired serial interface. All the information exchanges between the monitoring agent and the WSN goes through the bridge node. Using one of the nodes in the network as a bridge is convenient since it allows to leverage the flexibility of symbolic code execution on this node too as well as the other facilities of the development environment. The Symbolic Code Producer (SCP) automatically produces symbolic verification code. The SCP is a decision agent that takes as input the network structure and the verification rules related to a distributed application to appropriately concatenate snippets of symbolic code. The verification code makes the sensor node perform some application-specific computation and send the results back to the monitoring agent at one or more points during the execution of the application. Verification is thus carried out in virtue of this declared association between symbolic high-level code describing high-level operations and similarly written code verifying the outcome of the former. To this purpose, in the KB are defined rules of this type:

```
verific_code(Label, VerificCode)
```

in which `Label` specifies the operation to be verified, and `VerificCode` indicates the verification code to be transmitted by the monitoring agent to the bridge node so that its execution on arrival retrieve the desired results.

Once the SCP has produced the verification code, the CM starts the application execution by sending the initiating code to the network through the bridge node. Then it sends the verification code to the nodes accordingly to one of the monitoring modes.

Before the application execution begins, during the *initialization phase*, the SCP can inject executable code in the network targeting some or all the nodes. This is useful when debugging a WSN to ensure reproducibility of the performed tests by explicitly setting the configuration of the nodes, and to ensure that some preconditions are verified before the application execution. When the monitoring agent sends the initialization code to the network, it may also override some application-specific words to include debug functionalities. Moreover, this code can also be defined differently for each node. By sending a marker before the new definitions, at the end of the application execution, it is possible to revert the dictionary to its previous state.

The core element of the system is the Symbolic Code Verifier (SCV), which examines the results of the execution and ascertains the correctness of the application execution using the verification rules in the KB. Metrics about the distributed application execution and verification time as well as exchanged data are also recorded.

The verification process is detailed in Fig. 4. For the on demand and stepwise strategies, which are based on fine-grained evaluation during application execution, all nodes are queried for verification data at the end of the process.

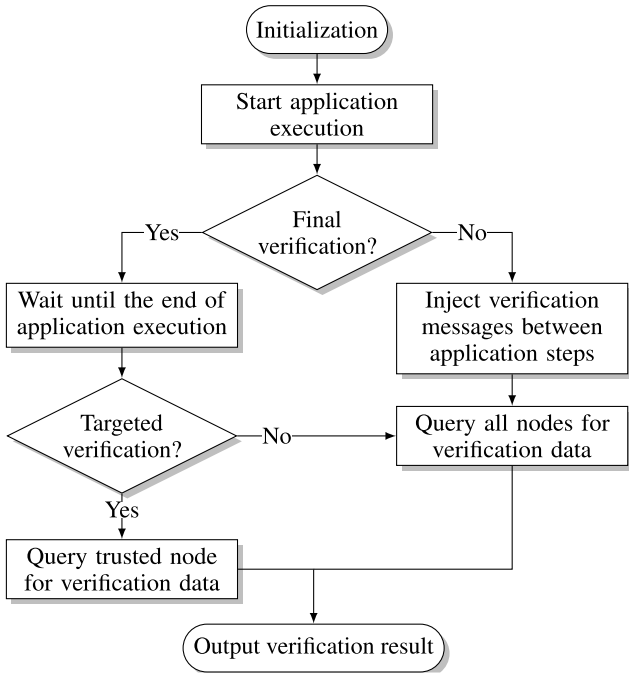


FIGURE 4. Flowchart of the verification process.

Remote nodes execute both application and verification code, once received. By executing the received verification code, networked devices collect the requested information and send back the actual results to the CM. Finally, results can be gathered and analyzed by the SCV subagent. In all the operating modes, the monitoring agent, through the SCV, goes through a decision-making process that takes into account assertions and facts in the knowledge base to automatically determine whether the application was executed correctly.

D. THE INFERENCE MODEL

According to the used monitoring mode, several verification rules in the knowledge base describe the relationships between the states of all the nodes in the network. These rules are appropriately selected by the monitoring agent to verify several post-conditions in specific moments during the execution of certain phases of the application, and eventually at its end, to ascertain whether the predicates describing the application state are satisfied. These predicates can be very flexible and of varying complexity. According to the application being tested, verification might entail ensuring that a variable in the remote nodes has a specific, predetermined, value, or it could need extensive reasoning that takes into consideration factors such as the topology of the network and the relationship between nodes. Moreover, the flexibility granted by the adopted symbolic approach and logic programming permits the verification of the correctness of some computation even if said computation cannot be reproduced locally. Verification rules can describe the relationship between values returned by the nodes without requiring the exact knowledge of what the computation result will be, either because of some stochastic component in the application or because the measurement of physical quantities is involved. Depending on the result of this verification process, the monitoring agent can take appropriate actions, such as stopping the verification process to report failure or testing the validity of additional predicates.

The KB allows for defining nodes as reliable, that is nodes whose response and behavior are assumed to be always correct. Rules defining primitives for communication, such as messages, and their storage and transmission among nodes, are also defined in the KB as well as routes and transmission timings.

The KB also models the distribution within the network and placement of nodes using a two-dimensional Cartesian coordinate system. The network structure is also modeled in the KB in terms of topology and connectivity as facts for each node in the KB.

Some of the predicates that can be used by the monitoring agent as conditions in its decision making process are shown in Table 1. These predicates take as input one or more node IDs and report some information regarding their status. This information can be used as conditions to determine the actions to pursue during the verification process. For instance, some nodes could be excluded from the verification process to avoid burdening nodes with low remaining charge or to respect a limit on the number of hops for the verification messages. Furthermore, in networks with heterogeneous devices, the available computational resources of each node and the available hardware peripheral can limit or expand the scope of the verification process.

TABLE 1. Examples of predicates that can be used as conditions by the monitoring agent.

`node_distance(X,Y,Distance)`	Topological information on the physical distance between nodes X and Y
`nodes_connected(X,Y,S)`	Information on whether nodes X and Y can communicate
`message_delivery_ratio(X,Y,Ratio)`	Information on the message delivery ration between nodes X and Y
`node_active(X,S)`	Information on whether node X is currently active
`is_node_bridge(X,S)`	Information on whether node X is the bridge node
`charge_level(X,S)`	Information on the charge level of the battery of node X
`is_node_reliable(X,S)`	Information on whether data from node X is deemed reliable
`computational_resources(X,S)`	Information on the available computational resources on node X
`available_sensors(X, SensorList)`	Information on the available sensors on node X
`sensor_status(X,Sensor,S)`	Information on the status of a specific sensor on node X

Once the queried values are retrieved from the nodes, the KB is enriched with additional information that the

TABLE 2. Examples of verification actions that can be performed by the monitoring agent.

`check_node_reply(X, Replies, S)`	Verifies that node X replied to verification messages
`compare_nth_value(N, Replies, S)`	Verifies that the n-th returned value is the same for every node
`match_nth_value(N, Reply, Value, S)`	Verifies that the n-th value returned by a node matches a given value
`is_in_range(N, Reply, Min, Max, S)`	Verifies that the n-th value returned by a node falls within a specified range of values
`verify_relationship(Reply_X, Reply_Y, R, S)`	Verifies that a specified relationship between the replies by two nodes holds

TABLE 3. Examples of verification rules that the monitoring agent uses to verify the correct application execution. If the condition `node_active` is false, no action is performed irrespectively of `sensor_status`.

Conditions		Actions	
`node_active`	`sensor_status`	`check_node_reply`	`is_in_range`
true	operative	✓	✓
true	defective	✓	×
false	*	×	×

monitoring agent can use in order to make decisions during the verification process. The predicates in Table 2 are examples of verification actions that might be performed by the monitoring agent to assess whether some conditions are satisfied by the values returned by the queried nodes. Specifically, the reported verification predicates assess if all the specified nodes replied to the verification message, and verify the validity of some relationships among the returned values.

Table 3 shows a possible decision rule that can be used by the monitoring agent to select the appropriate verification actions according to the status of a specific node. Given a node, if it is currently active the monitoring agent will check whether it replied to the verification messages, but the correctness of the returned values is only checked if a sensor on the node is not known to be defective. Inactive nodes are not queried and thus no verification action is performed on them, regardless of the sensor status. Verification rules can have an arbitrary number of combinations of conditions and actions and can involve multiple nodes. For the sake of brevity more complex rules are not presented.

Based on the symbolic code exchange mechanism, the CM sends application and verification code as plain text, without intermediate translation steps. Since words are interpreted by nodes, verification can be performed on-board by end devices during application execution, while the CM is responsible of collecting final results.

The same symbol may have different implementations on nodes to address hardware heterogeneity while still maintaining the same semantic meaning.

IV. SAMPLE APPLICATIONS

In order to test and validate the verification system, in this section, we present its adoption for developing some distributed applications of different complexity. For the sake of brevity, in the following we present some meaningful fragments together with the code provided for their verification. The proposed tool can monitor both code and network functionality. For the second application fragment only, which concerns network functionality, we describe in detail the logical reasoning process implemented for verification.

A. APPLICATION 1–AVERAGING

This application is a short fragment belonging to a more complex application for the distributed aggregation of physical quantities, in this specific case, instantiated to collect temperature data. The distributed application can be decomposed in the following steps:

1) the monitoring agent commands the bridge node to execute the symbolic code: `bcst tell: 0 0 update: tell`
   The `bcst` keyword specifies that the message be broadcast to the network, `update` is an application-specific word;
2) each listening node receives the message and executes it. As specified, the two zeros are interpreted as numeric values and put on the stack. The `update` symbol is defined to pick these two values from the stack and store them in two of its local variables. The first variable (`num`) holds the number of nodes that already carried out the temperature measurement. The second (`aggr`) holds the current aggregate temperature value, which is the sum of the measurements communicated by the nodes so far. The execution of the message thus commands the nodes to perform the initialization of the application by resetting their local values. The complete definition of `update`, which includes the `wait-and-reply` symbol that actually implements the rest of the distributed procedure (step 3), is:
   ```
   num ! aggr ! wait-and-reply
   ```
3) In order to synchronize the distributed execution, each node waits for a time proportional to its integer identifier. Subsequently, the node acquires the current temperature and updates the values of the two variables accordingly. Furthermore, the node updates the value of another local variable (`avg`) containing the average value of the temperatures just acquired. Then the node executes the symbolic code `bcst tell: <num> <aggr> update :tell`, that broadcasts the updated values of the variables `num` and `aggr`. All the other nodes, when receiving this message, execute the code updating the values of the two corresponding local variables, as in step 1.
4) The distributed execution terminates when all the network nodes have performed step 3. Knowing the node

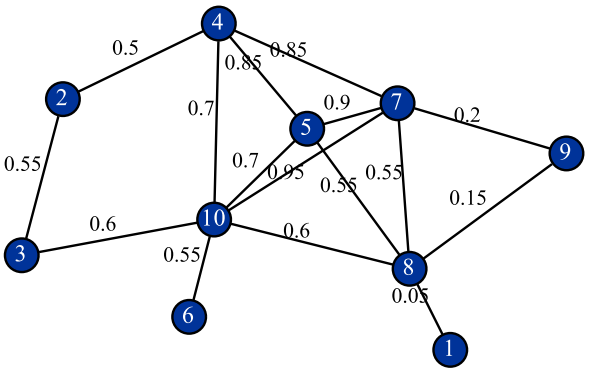


FIGURE 5. WSN graph with ratio of messages delivered in point-to-point transmissions shown on the connecting arcs.

identifier values, the monitoring agent can thus estimate when the distributed application has ended by comparing the elapsed time with the reply time of the node with the largest identifier value.

For the above distributed application, a simple verification consists in checking that all the network nodes converged to the same value and no update messages were lost. To this end, the following verification code is sent to the queried nodes:

```
num @ aggr @ avg @ reply tell:
                              ~ ~ ~ :tell
```

B. APPLICATION 2—NETWORK QUALITY MONITORING

The second sample application is a code portion of the network monitoring activity.

For this application, the KB holds information on the message delivery ratio for each pair of nodes in the network (Fig. 5), available through the `message_delivery_ratio` predicate shown in Table 1. However, a WSN may be deployed for an extended period and the connectivity characteristics may vary in time. The system monitors the current state of the network in order to plan more appropriate message routing. The procedure involves the following steps:

1) the bridge node broadcasts the initialization message:

```
0 net-quality
```

2) each listening node receives the initialization message and executes it. Again, the 0 is interpreted as a numeric value and put on the stack. The `net-quality` symbol is defined to either reset the counter of received messages or to increment it if the value on top of the stack is either 0 or 1 respectively (`reset-or-increment` symbol). The counter is held in the `rcvd-msg` variable. In this step the counters of all the listening node are thus reset. Finally, the node start executing the next step (3) defined in the `wait-and-reply-nq` symbol as it can be seen in the definition of `net-quality`:

```
reset-or-increment wait-and-reply-nq
```

3) The `wait-and-reply-nq` symbol starts a timer letting the node idle for a time proportional to its ID. When the timer expires, the node broadcasts the update message:

```
1 net-quality
```

4) as described before, the definition of `net-quality` is such that whenever a node receives the above message from any other node, it increments its own counter.

Once the application execution is terminated, the monitoring agent starts the verification process. In order to monitor the connectivity of the network, verification is performed on the number of messages received by each node. To this end, the SCP will select the appropriate snippet of verification code for this application. According to Table 4 the agent will use the"`rcvd-msg @`" snippet to extract the required counter value from each node.

TABLE 4. Predicates in the KB to associate applications and verification code snippets.

```
verific_code(network-quality, "rcvd-msg @")
verific_code(averaging, "num @ aggr @ avg @")
verific_code(network-discovery, "leaf @ parent @")
verific_code(hvac-control, "temperature @ dup
classify HVAC-signal @")
```

To make each node report the number of received messages, the SCP concatenates the appropriate communication primitives. After the verification code snippet, a `tell` construct is used to send back the computed values. For every node to be verified, the SCP merges the verification code into a `tell` construct addressed to the correct destination node. The code that the CM sends to the bridge node in order to query a node is the following:

```
NodeID tell: rcvd-msg @ reply tell:
                              ~ :tell  :tell
```

with `NodeID` the address of the node the verification code is sent to.

If the `nodes_connected(bridge, NodeID)` condition is not satisfied, meaning that in the KB there is no information reporting a direct link between the bridge and the target node, the SCP selects the `forward` construct instead of the `tell` communication primitive to ensure that messages can be correctly delivered.

Once the nodes are queried, the SCV performs verification actions for all the nodes that satisfy the `node_active` condition in Table 1 through the following predicates:

```
findall(X, node_active(X,true),
     ActiveNodes),
     verify_nodes(network-quality,
     ActiveNodes, Replies)
```

The monitoring agent, exploiting its knowledge of the connection graph and the previous estimated message delivery rate, can compare the received values with its estimation of the average number of messages that each node should receive. The monitoring agent can perform different verification actions depending on whether all node pairs have a 100% message reception rate.

If each link in the network is lossless, at the end of the application execution the i-th node will receive N_i messages, with N_i the number of nodes satisfying the `nodes_connected` condition with node i. The counter of node i will then hold said value. As an extreme case, in a network topology where all nodes can communicate directly, all counters will show the same value.

When not all node pairs have a 100%, the reception of each message by a specific node is a Bernoulli trial, with chances of success depending on the message source. The total amount of messages received by a node, then, is the outcome of a series of independent Bernoulli trials with different distributions, and it can be modeled as a Poisson binomial distribution. Accordingly, each node has a different probability of receiving messages from a source, hence they can have different expected values.

In case links are lossless the monitoring agent selects the `match_nth_value` verification action from Table 2 with value N. Otherwise, verification is performed using the expected number of received messages computed for each node. Moreover, if the stochastic model is used, the monitoring agent can also perform the `is_in_range` verification action, using the computed expected value and standard deviation to obtain a range.

As previously stated, in this example the verification tool is only used to monitor variations in the status of the hardware.

For the sake of brevity, only a brief overview of the application and the verification actions are described in the other sample applications.

C. APPLICATION 3—NETWORK DISCOVERY

The third sample application is based on the network discovery protocol described in [45] and used to construct a network topology tree. The application execution entails the following steps:

1) the bridge node starts the application by broadcasting the following code:

```
-1 network-discovery
```

   every node within communication range of the bridge node will receive this message;
2) each receiving node will record the source of the message as its parent node in the topology tree and starts a timer proportional to its ID;
3) when the timer expires, the node broadcasts the ID of its parent node by executing the following code:

```
parent @ bcst tell:
        ~ network-discovery :tell
```

4) one of three cases can happen when a node receives this message:
   a) the node is the broadcast parent node: the receiving node records that it is not a leaf node in the topology tree;
   b) the node currently has no parent node: in this case the receiving node starts executing the distributed application from step 2;
   c) in all the other circumstances the message is ignored.
5) the distributed application terminates when all the network nodes have performed step 3.

For this application, the test is about whether each node computed its correct position on the network topology tree. To this end, the following verification code is sent to the queried nodes:

```
leaf @ parent @ reply tell: ~ ~ :tell
```

D. APPLICATION 4—HVAC CONTROL

In this last sample application we face some issues related to non-determinism. An appropriate monitoring of non-deterministic behaviors is a crucial issue since many applications, for instance those of IoT systems, are often characterized by uncertainties. IoT devices typically execute specific actions on the basis of their sensor readings. However, measurements can be inaccurate or actuators might malfunction. Furthermore, these applications are characterized by the unpredictability of message exchanges. Finally, these applications might not necessarily terminate after a definite time as, for instance, when implementing control loops for physical processes. All these considerations underline the difficulty of monitoring the correct behavior of IoT applications while minimizing undue interference. For these reasons, the stepwise monitoring mode, as described in Section III-B, can be a valuable tool.

The following application fragment implements a temperature control in a smart environment.

1) On startup all the nodes acquire a temperature sample;
2) Each node, acquired its first sample, classifies it as: 1) belonging to the user predefined comfort range, 2) cold, or 3) warm, and broadcasts the classification value; this step is encoded as such:

```
temperature @ classify bcst tell:
        ~ temperature-update :tell
```

3) All the nodes periodically perform the same measurements. Only when the classification of new acquired temperature is different from the previous one, the nodes broadcast a message with the new classification;
4) According to the classification emerging from the majority of nodes, a collector node sends opportune commands to the HVAC system so to guarantee that most of the nodes obtain readings in the comfort range.

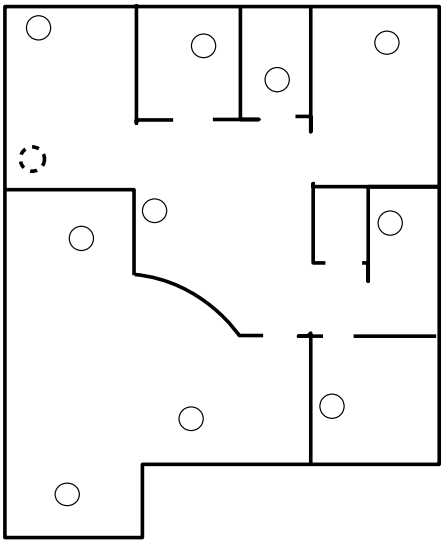


FIGURE 6. The home environment used for the tests. Circles represent nodes. The node acting as a bridge to the host is indicated with a dashed circled.

The verification of correct functioning entails querying all nodes for the latest temperature sample together with its classification and retrieving the signal sent to the HVAC system in order to assess whether the temperatures are classified correctly and the current HVAC setting is addressing the requirements of most of the nodes.

In order to acquire the verification data, the following symbolic code is sent to the queried nodes:

```
temperature @ dup classify HVAC-signal
              @ reply tell: ~ ~ ~ :tell
```

V. EXPERIMENTAL EVALUATION

We evaluated our approach by performing experiments on three 10-node WSNs differently arranged to provide a representative sample of real topologies. The first network setting was a linear topology with each node able to communicate only with the two closest peers. The second network was arranged in an L-shaped topology with the bridge placed in the middle of the segment connecting the two extreme nodes. The third network was built by deploying nodes in a home environment (Fig. 6). In the latter, two networks the bridge was able to exchange messages with any node in one hop. In all the arrangements, each node was a Crossbow IRIS mote equipped with an IEEE 802.15.4 compliant radio transceiver, an Atmega1281 8-bit Harvard RISC 16 MHz processor, 128 KB of Flash memory, 8 KB of static RAM, and a 4 KB EEPROM provided with sensors to read physical quantities such as temperature, light, and ambient noise.

For every combination of application, topology, and verification strategy, fifteen runs of verification were carried out, measuring the number of exchanged messages, the bytes sent through serial line to the bridge node, and the average required time. Since for most of the tested applications the execution time depends on the node IDs, the ability of the monitoring agent to virtualize IDs was exploited to randomly assign MAC values in the range 0–65535. Being the ID randomization functionality based on a fixed pool of seeds, each combination was evaluated on the same fifteen sets of IDs. The assigned IDs were set on the nodes through executable code sent by the verification system in the initialization phase. The overhead from these operations does not contribute to verification time measurements because an already deployed network does not generally require these steps.

In the linear topology setting, randomizing the addresses presented an extra challenge. Before each experiment the routing tables in the whole network required to be set up with the randomized addresses. Due to the flexibility of the system, this was simply solved by having the monitoring agent send each node executable code defining its routing table in the initialization phase, before the start of the application. To ease the practicality of the experiments, during the initialization phase the nodes were gathered in close proximity to ensure that a correct configuration was quickly achieved. With no loss of generality regarding the evaluation of the overhead introduced by the monitoring agent, the topology was though linear from the point of view of the verification system.

Table 5 collects the results of all the tests:

- t_p is the time elapsed from the moment the message starting application execution is sent to the moment the application terminates and global verification can begin. It can be computed by the system before starting the application and is not influenced by the verification modality. It does not apply to the stepwise modality.
- $t_p + t_v$ is the time from the moment the message starting the application execution is sent to the verification end.
- the *messages* column reports the number of exchanged messages during application execution and verification. The messages considered are only those needed to verify the application itself, not those exchanged by the nodes during the normal application execution, as those are not related to the verification tool. In multi-hop topologies each message forwarding is counted separately.
- the *bytes* column reports the bytes sent through serial line to the bridge node

The *targeted* strategy, interrogating only a reliable remote node, is the most efficient. Nevertheless, besides the intrinsic difficulty of selecting a subset of the nodes as reliable, especially for long executions, not every application can be verified with knowledge about the state of a single node.

The *global* strategy can ensure that the application produce the correct final result but at the cost of higher verification time and number of exchanged messages. Moreover, if the final result is not correct, it may not be possible to determine what caused the failure.

The *on demand* strategy entails even more exchanged messages and may potentially introduce timing issues. However, this strategy provides more detailed information during the application execution, for instance about error conditions, which could be detected before the end of the process.

TABLE 5. Experimental results.

Modality	Linear				L-shaped				Home			
Averaging												
	$t_p[s]$	$t_p + t_v[s]$	messages	bytes	$t_p[s]$	$t_p + t_v[s]$	messages	bytes	$t_p[s]$	$t_p + t_v[s]$	messages	bytes
global		291.09	111.00	1112.60		278.19	21.00	1004.60		272.90	21.00	1004.60
targeted	149.70	166.43	15.00	152.93	149.70	165.60	3.00	140.93	149.70	165.06	3.00	140.93
on demand		336.14	150.33	1463.60		317.72	27.53	1318.00		309.89	27.53	1318.00
stepwise	-	294.28	221.00	2324.20	-	269.36	41.00	2108.20	-	271.27	41.00	2108.20
Net quality monitoring												
global	149.83	283.13	111.00	1044.93	149.83	251.69	21.00	867.60	149.83	253.69	21.00	867.60
Net disc												
global		1066.22	111.00	1096.60		418.11	21.00	988.60		419.39	21.00	988.60
targeted	933.98	952.10	15.00	172.93	296.17	313.03	3.00	160.93	296.17	313.62	3.00	160.93
on demand		1173.93	196.20	1978.73		447.94	27.00	1264.40		454.44	27.00	1264.40
HVAC control												
stepwise	-	294.51	183.00	2430.00	-	282.20	41.00	2300.93	-	281.13	41.00	2283.73

The *stepwise* scheme, by essentially pausing the application execution at each step, enables the possibility of monitoring the internal state of all the nodes; this capability can be leveraged to obtain fine-grained information. This strategy, however, has the highest intrusion on the WSN and alters the normal network behavior.

From the $t_p + t_v$ column, in the averaging application, it can be seen that the stepwise verification modality reduces the waiting time with respect to the on demand modality because the t_p component is dominated by idle waiting times. An application with more frequent messages exchanges would be slowed down.

The HVAC control application was only tested in the stepwise modality because of its non-deterministic nature. The non-deterministic update mechanism does not allow for determining safe timings to send messages without collisions. For the sake of ease of testing, temperature readings were simulated.

The network quality monitoring application is meant to assess the state of the whole network, for this reason the targeted modality makes little sense. Moreover, verification is performed on the final results of the application so we did not perform the on demand and stepwise verifications.

The relationship between the number of queries performed by the rule system and the other tracked metrics is reported in Fig. 7. As expected, the chart shows a noticeable linear correlation between the number of queries required for verification and the time spent to perform it.

The number of messages propagated through the network, on the other hand, shows high variability between the two 1-hop topologies (L-shaped and Home) and the linear one: in the latter, for each message sent from the monitoring agent several messages are generated by the nodes contributing to the overall message count. In the stepwise verification scheme for the HVAC control application the non-deterministic execution order of the nodes makes the number of exchanged messages highly variable.

Despite the noticeable influence of topology and application in the number of messages propagated through the network, a noticeable linear correlation between the number of queries performed by the monitoring agent and the t_v component of the verification time can be observed. In fact, the

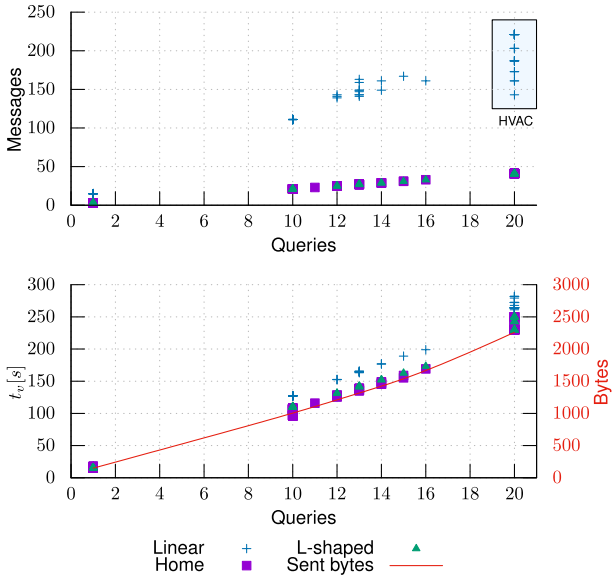


FIGURE 7. Relationship between t_v and the number of queries with varying topologies for all applications and verification schemes. The verification time mainly depends from the number of queries even if the generated messages may vary substantially depending on topology and tested application.

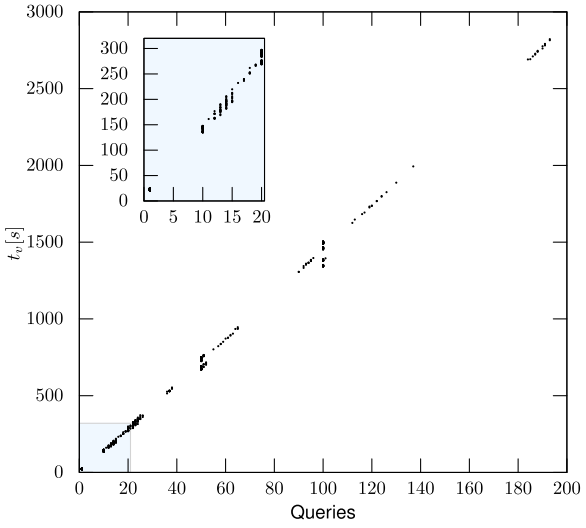


FIGURE 8. The linear relationship between the number of queries and t_v roughly holds even with an increased network size. The highlighted area in the chart contains the results of the simulations with the same network size of the tests on deployed WSNs.

main bottleneck is not the transmission of messages among nodes but the 9600 bit/s rate of the serial connection between the CM and the bridge node: dynamic code execution enables the propagation of a message through the network with little extra burden. The amount of code sent from the CM to the bridge to be dispatched to the network for a query is similar in all the conditions as shown in the rightmost chart.

Higher bitrates with such constrained resources, would not be feasible. This is not a limitation of the monitoring agent.

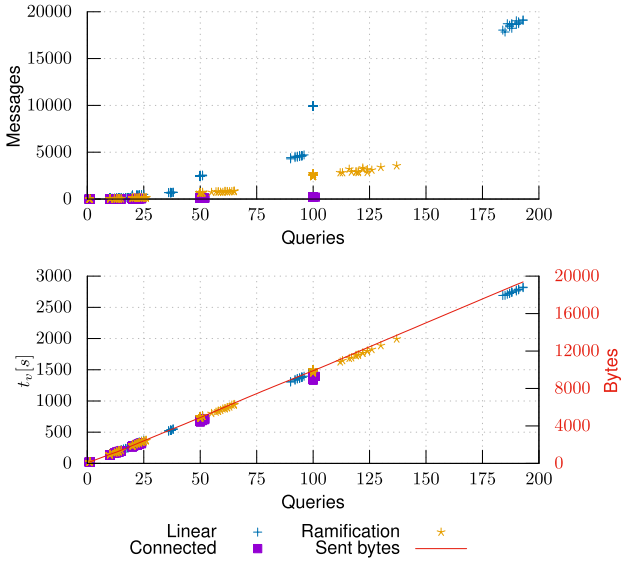


FIGURE 9. Compared with the number of messages that need to travel through the network, verification time depends almost exclusively from the number of queries.

In fact, the actual throughput would be limited by the low computational power of the nodes and their tiny amounts of RAM for buffers that would trigger control-flow mechanisms anyway. All in all, the fact that our approach is feasible even when targeting such a resource-poor platform shows its effectiveness.

To asses the applicability of the proposed methodology in networks with more nodes we also performed numerical simulations for each verification scheme. The simulations were carried out with multiple network topologies:

- connected topologies where the bridge node could directly query each node;
- linear topologies with the bridge node at one end of the line;
- ramified topologies where each node could directly communicate with at least four other nodes.

The networks were generated in different sizes: 10, 20, 50, and 100 nodes. Fig. 8 reports t_v and performed queries for all the performed simulations showing the linear relationship: each query to a node had a cost of $\sim$ 15 s. The simulation results for networks of size 10 closely match the tests performed on the deployed networks.

Fig. 9 summarizes the simulation results, and shows that the previously identified relationships hold even at increased network size. In particular, the impact of the transmissions of the generated messages on t_v is negligible when compared to the communication through serial line with the bridge node. Since the rate at which queries were performed was far lower than the maximum throughput of the network, the verification process minimally interfered with the WSN operations. Moreover, the number of bytes sent through the serial interface for each query was constant, thus the burden on the bridge node was the same regardless of the topology and the application under test.

Results also show that verification time does not depend on network topology but only on the number of queries performed. This finding may be used to develop more advanced verification schemes that only select a subset of nodes to query in order to comply with constraints time available for verification. In addition to the specific ability of the tool to test distributed applications on deployed WSNs, its viability is also supported by the linear scaling of verification time with respect to the number of queries and, by extension, the number of nodes, the capability of performing verification operations in different topologies, and the absence of any need of special-purpose debugging hardware or software.

VI. CONCLUSION

In this paper, a system supporting modeling and verification of distributed applications running on WSN nodes was introduced. The system is based on a symbolic programming paradigm that enables modeling applications with a high level of abstraction even on resource-constrained devices.

The core feature of the proposed system is represented by the use of logical reasoning to assess network functionality and code correctness. The functionalities of the monitoring agent have also been shown through four running examples of application fragments.

The experimental results concerning several applications in multiple deployed networks and extensive simulations support the feasibility of the approach to test distributed applications running on resource-constrained WSN nodes as the time spent in application verification was comparable to application execution time.

Future work will consider running the experiments on virtualized nodes alongside physical ones and verification of hybrid simulations of distributed applications: taking advantage of the easy reconfigurability of the nodes, a node may be automatically reconfigured during the application execution to act as a different node in the simulated network, enabling complex analyses on relatively small networks.

REFERENCES

[1] A. Khanna and S. Kaur, "Internet of Things (IoT), applications and challenges: A comprehensive review," *Wireless Pers. Commun.*, vol. 114, no. 2, pp. 1687–1762, Sep. 2020.

[2] M. Assim and A. Al-Omary, "Design and implementation of smart home using WSN and IoT technologies," in *Proc. Int. Conf. Innov. Intell. Informat., Comput. Technol. (3ICT)*, Dec. 2020, pp. 1–6.

[3] Y. A. Qadri, A. Nauman, Y. B. Zikria, A. V. Vasilakos, and S. W. Kim, "The future of healthcare Internet of Things: A survey of emerging technologies," *IEEE Commun. Surveys Tuts.*, vol. 22, no. 2, pp. 1121–1167, 2nd Quart., 2020.

[4] A. De Paola, P. Ferraro, S. Gaglio, G. Lo Re, M. Morana, M. Ortolani, and D. Peri, "An ambient intelligence system for assisted living," in *Proc. AEIT Int. Annu. Conf.*, Sep. 2017, pp. 1–6.

[5] S. Ramnath, A. Javali, B. Narang, P. Mishra, and S. K. Routray, "IoT based localization and tracking," in *Proc. Int. Conf. IoT Appl. (ICIOT)*, May 2017, pp. 1–4.

[6] R. C. Shit, S. Sharma, D. Puthal, and A. Y. Zomaya, "Location of things (LoT): A review and taxonomy of sensors localization in IoT infrastructure," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 3, pp. 2028–2061, 3rd Quart., 2018.

[7] S. Neelakandan, M. A. Berlin, S. Tripathi, V. B. Devi, I. Bhardwaj, and N. Arulkumar, "IoT-based traffic prediction and traffic signal control system for smart city," *Soft Comput.*, vol. 25, no. 18, pp. 12241–12248, Sep. 2021.

[8] V. Kuppusamy, U. Thanthrige, A. Udugama, and A. Förster, "Evaluating forwarding protocols in opportunistic networks: Trends, advances, challenges and best practices," *Future Internet*, vol. 11, no. 5, p. 113, May 2019.

[9] A. El-Mouaffak and A. E. B. El Alaoui, "Considering the environment's characteristics in wireless networks simulations and emulations: Case of popular simulators and WSN," in *Proc. 3rd Int. Conf. Netw., Inf. Syst. Secur.*, Mar. 2020, pp. 1–4.

[10] E. Weingartner, H. vom Lehn, and K. Wehrle, "A performance comparison of recent network simulators," in *Proc. IEEE Int. Conf. Commun.*, Jun. 2009, pp. 1–5.

[11] E. Volnes, S. Kristiansen, and T. Plagemann, "Improving the accuracy of timing in scalable WSN simulations with communication software execution models," *Comput. Netw.*, vol. 188, Apr. 2021, Art. no. 107855.

[12] C. Adjih, E. Baccelli, E. Fleury, G. Harter, N. Mitton, T. Noel, R. Pissard-Gibollet, F. Saint-Marcel, G. Schreiner, J. Vandaele, and T. Watteyne, "FIT IoT-LAB: A large scale open experimental IoT testbed," in *Proc. IEEE 2nd World Forum Internet Things (WF-IoT)*, Dec. 2015, pp. 459–464.

[13] C. A. Boano, T. Voigt, C. Noda, K. Römer, and M. Zúñiga, "JamLab: Augmenting sensornet testbeds with realistic and controlled interference generation," in *Proc. 10th ACM/IEEE Int. Conf. Inf. Process. Sensor Netw.*, 2011, pp. 175–186.

[14] J. Albesa, R. Casas, M. T. Penella, and M. Gasulla, "REALnet: An environmental WSN testbed," in *Proc. Int. Conf. Sensor Technol. Appl. (SENSORCOMM)*, Oct. 2007, pp. 502–507.

[15] L. P. Steyn and G. P. Hancke, "A survey of wireless sensor network testbeds," in *Proc. IEEE Africon*, Sep. 2011, pp. 1–6.

[16] L. Sanchez, L. Muñoz, J. A. Galache, P. Sotres, J. R. Santana, V. Gutierrez, R. Ramdhany, A. Gluhak, S. Krco, E. Theodoridis, and D. Pfisterer, "SmartSantander: IoT experimentation over a smart city testbed," *Comput. Netw.*, vol. 61, pp. 217–238, Mar. 2014.

[17] P. Appavoo, E. K. William, M. C. Chan, and M. Mohammad, "Indriya2: A heterogeneous wireless sensor network (WSN) testbed," in *Proc. Int. Conf. Testbeds Res. Infrastruct.* Cham, Switzerland: Springer, 2018, pp. 3–19.

[18] P. Hurni, M. Anwander, G. Wagenknecht, T. Staub, and T. Braun, "TARWIS—A testbed management architecture for wireless sensor network testbeds," in *Proc. 7th Int. Conf. Netw. Service Manage.*, 2011, pp. 1–4.

[19] X. Xian, W. Shi, and H. Huang, "Comparison of OMNET++ and other simulator for WSN simulation," in *Proc. 3rd IEEE Conf. Ind. Electron. Appl.*, Jun. 2008, pp. 1439–1443.

[20] U. M. Colesanti, C. Crociani, and A. Vitaletti, "On the accuracy of OMNET++ in the wireless sensornetworks domain: Simulation vs. testbed," in *Proc. 4th ACM Workshop Perform. Eval. Wireless Sensor, Ubiquitous Netw.*, 2007, pp. 25–31.

[21] P. Nayak and P. Shree, "Comparison of routing protocols in WSN using NetSim simulator: LEACH vs LEACH-C," *Int. J. Comput. Appl.*, vol. 106, no. 11, pp. 1–6, 2014.

[22] S. Saginbekov and C. Shakenov, "Hybrid simulators for wireless sensor networks," in *Proc. IEEE Conf. Wireless Sensors (ICWiSE)*, Oct. 2016, pp. 59–65.

[23] G. Z. Papadopoulos, A. Gallais, G. Schreiner, E. Jou, and T. Noel, "Thorough IoT testbed characterization: From proof-of-concept to repeatable experimentations," *Comput. Netw.*, vol. 119, pp. 86–101, Jun. 2017.

[24] H. Yetgin, K. T. K. Cheung, M. El-Hajjar, and L. H. Hanzo, "A survey of network lifetime maximization techniques in wireless sensor networks," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 2, pp. 828–854, 2nd Quart., 2017.

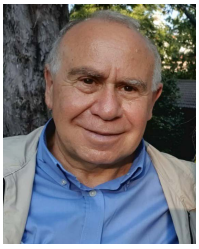
[25] Y. Gao, F. Xiao, J. Liu, and R. Wang, "Distributed soft fault detection for interval type-2 fuzzy-model-based stochastic systems with wireless sensor networks," *IEEE Trans. Ind. Informat.*, vol. 15, no. 1, pp. 334–347, Jan. 2019.

[26] A. Schoofs, G. M. P. O'Hare, and A. G. Ruzzelli, "Debugging low-power and lossy wireless networks: A survey," *IEEE Commun. Surveys Tuts.*, vol. 14, no. 2, pp. 311–321, 2nd Quart., 2012.

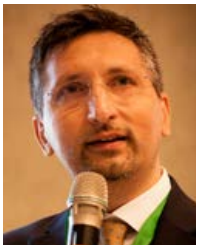
[27] J. V. Capella, J. C. Campelo, A. Bonastre, and R. Ors, "A reference model for monitoring IoT WSN-based applications," *Sensors*, vol. 16, no. 11, p. 1816, Oct. 2016.

[28] S. Gaglio, G. Lo Re, G. Martorella, and D. Peri, "DC4CD: A platform for distributed computing on constrained devices," *ACM Trans. Embedded Comput. Syst.*, vol. 17, no. 1, pp. 27:1–27:25, 2017.
[29] G. Tolle and D. Culler, "Design of an application-cooperative management system for wireless sensor networks," in *Proc. 2nd Eur. Workshop Wireless Sensor Netw.*, 2005, pp. 121–132.
[30] M. Hanninen, J. Suhonen, T. D. Hamalainen, and M. Hannikainen, "Practical monitoring and analysis tool for WSN testing," in *Proc. Conf. Design Architectures Signal Image Process. (DASIP)*, Nov. 2011, pp. 1–8.
[31] M. N. Mendoza, J. C. C. Rivadulla, A. M. B. Pina, J. V. C. Hernandez, and R. O. Carot, "HMP: A hybrid monitoring platform for wireless sensor networks evaluation," *IEEE Access*, vol. 7, pp. 87027–87041, 2019.
[32] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse, "Clairvoyant: A comprehensive source-level debugger for wireless sensor networks," in *Proc. 5th Int. Conf. Embedded Netw. Sensor Syst.*, 2007, p. 189.
[33] M. M. Mozumdar, W. Bian, J. Perez, and L. Lavagno, "RSD-WSN: Remote source-level debugger for rapid application development in wireless sensor networks," in *Sensors and Systems for Space Applications VI*, vol. 8739, P. Khanh, C. Joseph, H. Richard, and C. Genshe, Eds. Bellingham, WA, USA: SPIE, 2013, pp. 266–271.
[34] Y. He and S. Lian, "Stethoscope: A sustainable runtime debugger for wireless sensor networks," in *Proc. IEEE 21st Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2015, pp. 250–257.
[35] Y.-J. Kim, S. Song, and D. Kim, "HDF: Hybrid debugging framework for distributed network environments," *ETRI J.*, vol. 39, no. 2, pp. 222–233, Apr. 2017.
[36] S. Gaglio, G. Lo Re, G. Martorella, and D. Peri, "A fast and interactive approach to application development on wireless sensor and actuator networks," in *Proc. Emerg. Technol. Factory Automat. (ETFA)*, 2014, pp. 1–8.
[37] A. Augello, R. D'Antoni, S. Gaglio, G. Lo Re, G. Martorella, and D. Peri, "Verification of symbolic distributed protocols for networked embedded devices," in *Proc. 25th IEEE Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, Sep. 2020, pp. 1177–1180.
[38] S. Gaglio, G. Lo Re, G. Martorella, and D. Peri, "Knowledge-based verification of concatenative programming patterns inspired by natural language for resource-constrained embedded devices," *Sensors*, vol. 21, no. 1, p. 107, Dec. 2020.
[39] S. Gaglio, G. Lo Re, G. Martorella, and D. Peri, "WSN design and verification using on-board executable specifications," *IEEE Trans. Ind. Informat.*, vol. 15, no. 2, pp. 710–718, Feb. 2018.
[40] J. Baek, S. I. Han, and Y. Han, "Optimal UAV route in wireless charging sensor networks," *IEEE Internet Things J.*, vol. 7, no. 2, pp. 1327–1335, Feb. 2020.
[41] J. S. Furter and P. C. Hauser, "Interactive control of purpose built analytical instruments with forth on microcontrollers—A tutorial," *Analytica Chim. Acta*, vol. 1058, pp. 18–28, Jun. 2019.
[42] M. Bošnjak, T. Rocktäschel, J. Naradowsky, and S. Riedel, "Programming with a differentiable forth interpreter," in *Proc. 34th Int. Conf. Mach. Learn.*, vol. 70, P. Doina and T. Y. Whye, Eds. Sydney, NSW, Australia, 2017, pp. 547–556.
[43] M. Tancreti, V. Sundaram, S. Bagchi, and P. Eugster, "TARDIS: Software-only system-level record and replay in wireless sensor networks," in *Proc. 14th Int. Conf. Inf. Process. Sensor Netw.*, Apr. 2015, pp. 286–297.
[44] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975.
[45] S. Gaglio, G. Lo Re, G. Martorella, and D. Peri, "A lightweight network discovery algorithm for resource-constrained IoT devices," in *Proc. Int. Conf. Comput., Netw. Commun. (ICNC)*, Feb. 2019, pp. 355–359.

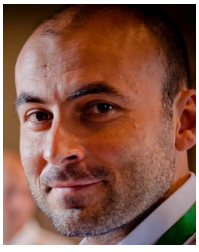
ANDREA AUGELLO received the bachelor's and master's degrees in computer engineering from the University of Palermo, Palermo, Italy, in 2019 and 2021, respectively, where he is currently pursuing the Ph.D. degree with the Department of Engineering. His current research interests include distributed systems and artificial intelligence.



SALVATORE GAGLIO (Life Member, IEEE) received the Graduate degree in electronic engineering from the University of Genoa, Genoa, Italy, in 1977, and the M.S.E.E. degree from the Georgia Institute of Technology, Atlanta, USA, in 1978. Since 1986, he has been a Professor in computer science and artificial intelligence with the University of Palermo, Italy, where he has been the scientific responsible of the Ph.D. program in computer engineering, since 2007. From 2015 to 2018, he was the President of the Italian National Academy of Sciences, Humanities, and Arts of Palermo. His current research interests include artificial intelligence and robotics. He has been a member of various committees for projects of national interest in Italy and he is referee of various scientific congresses and journals. He is a member of ACM and AAAI.



GIUSEPPE LO RE (Senior Member, IEEE) received the Laurea degree in computer science and the Ph.D. degree in computer engineering from the University of Pisa, in 1990 and 1999, respectively. He is currently a Full Professor in computer engineering with the University of Palermo. His current research interests include the area of computer networks and distributed systems, focusing on reputation and security systems. He is a Senior Member of IEEE Communication Society and the Association for Computer Machinery.



DANIELE PERI received the master's and Ph.D. degrees in computer engineering from the University of Palermo, Palermo, Italy, in 1998 and 2004, respectively. He is currently an Assistant Professor in computer engineering with the University of Palermo. His current research interests include intelligent and distributed embedded systems, embedded architectures, system software design, symbolic and logic programming, and specification and verification of embedded systems.

Open Access funding provided by 'Università degli Studi di Palermo' within the CRUI CARE Agreement