Article

A Formal Methodology to Design and Deploy Dependable Wireless Sensor Networks

Juan Carlos Augusto [1], Marcello Cinque [2], Antonio Coronato [3]* and Alessandro Testa [4]

1 Middlesex University of London, UK; j.augusto@mdx.ac.uk
2 University of Naples "Federico II", Italy; macinque@unina.it
3 CNR-ICAR, Italy; antonio.coronato@icar.cnr.it
4 Ministero dell'Economia e delle Finanze, Italy; alessandro.testa@tesoro.it
* Correspondence: antonio.coronato@icar.cnr.it; Tel.: +39-081-6139-511

Abstract: Wireless Sensor Networks (WSNs) are being increasingly adopted in critical applications, where verifying the correct operation of sensor nodes is a major concern. Undesired events may undermine the mission of the WSNs. Hence their effects need to be properly assessed before deployment to obtain a good level of expected performance and during the operation in order to avoid dangerous unexpected results. In this paper we propose a methodology to support design and deployment of dependable WSNs by means of an event-based formal verification technique. The methodology includes a process to guide designers towards the realization of a dependable WSN and a tool ("ADVISES") to simplify its adoption. The tool allows to generate automatically formal specifications used to check correctness properties and evaluate dependability metrics at design time and at runtime. During the runtime we can check the behavior of the WSN accordingly to the results obtained at design time and we can detect sudden and unexpected failures, in order to trigger recovery procedures. The effectiveness of the methodology is shown in the context of two case studies, aiming to illustrate how the tool is helpful to drive design choices and to check the correctness properties of the WSN at runtime.

Keywords: Wireless Sensor Networks; Formal Methods; Dependability; Metrics; Modeling

1. Introduction & Motivation

Wireless Sensor Networks (WSNs) [1] are being increasingly used into critical application scenarios where the level of trust on WSNs becomes an important factor, affecting the success of industrial WSN applications. The extensive use of this kind of networks stresses the need to verify their dependability [1] not only at design time to prevent wrong design choices but also at runtime in order to make a WSN more robust against failures that may occur during its operation.

Typical critical scenarios are environmental monitoring (e.g. detection of fires in forests [1]), structural monitoring of civil engineering structures [3], health monitoring (in medical scenarios) [4] and patient monitoring [5][6] by means of Ambient Intelligence (AmI[7]) systems. For example, in the case of remote patient monitoring, alerts must be raised and processed within a temporal threshold; functional incorrectness or runtime failures may result in catastrophic consequences for the patient. These types of applications are considered safety-critical and they must be designed and developed with intrinsic and stringent dependability requirements [8]. Thus, depending on the application scenarios, different dependability requirements can be defined, such as, node lifetime, network resiliency and coverage of the monitoring area.

The work presented in [9] evidenced that also in a simple deployment, a single node can be responsible of the failure of a huge piece of the network. For instance, a node that is close to the sink

[1] Dependability is defined as the ability of a system to deliver a service that can justifiably be trusted [2]. It is an integrated concept encompassing the attributes of reliability, availability, maintainability, safety, and integrity.

(i.e., the gateway of the WSN that has the role to collect all of the measures detected by the sensors) is more likely to fail due to the great demand it is subjected to, and its failure would likely cause the isolation of a set of nodes from the sink.

Therefore it is necessary to verify the WSNs at design time against given correctness properties, in order to increase the confidence about the robustness of the designed solution before putting it into operation. Moreover, it is also important to avoid unexpected results or dangerous effects during the runtime of the WSN; this can be obtained by checking traces of events generated from the system run against the same correctness properties used at design time.

Formal methods can be used for these purposes, due to their wide adoption in the literature to verify the correctness of a system specification [10] or to perform runtime verification [11] [12]. However, the practical use of formal methods for the verification of dependability properties of WSNs has received little attention, due to the distance between system engineers and formal methods experts and the need to re-adapt the formal specification to different design choices. Even if a development team would invest on the definition of a detailed specification of WSN correctness properties, a design change (e.g., different network topology) could require to rethink the formal specification, incurring in extra undesirable costs.

To overcome these limitations, the contribution of this paper is manyfold. Specifically, we propose:

- a formal methodology to support the design and deployment of dependable WSNs both at design time (*static verification*) and at runtime (*runtime verification*);
- the definition of a unique formal specification of WSN correctness, based on the event calculus formalism, subdivided in two logical subsets: a *general correctness specification*, valid independently of the particular WSN under study, and a *structural specification* related to the properties of the target WSN (e.g., number of nodes, topology, etc.);
- the adoption of specific WSN dependability metrics, such as *connection resiliency* and *coverage*, introduced in [13], for measuring dependability degree and having a quantitative assessment of a WSN;
- an automated verification tool, named *ADVISES* (**A**utomate**D** **V**er**I**fication of w**S**n with **E**vent calculu**S**), to facilitate the adoption of the proposed approach.

The key idea of the proposed methodology is to base the verification of correctness properties on a set of specifications that can be used interchangeably at design time and at runtime. The decomposition of the specification in two sets simplifies the adoption of the approach. While general correctness specifications do not need to be adapted when changing the target WSN, structural specifications depend on the target, and are designed to be generated automatically.

The ADVISES tool facilitates the adoption of the proposed methodology by system engineers with no experience on formal methods. At design time, it can be used to perform a *robustness checking* of the target WSN, i.e., to verify the long term robustness of the WSN (in terms of the proposed metrics) against random sequences of undesired events, useful to identify corner cases and dependability bottlenecks. At runtime, it monitors the deployed WSN. If an undesired event occurs, the tool calculates the current values of dependability metrics (e.g. raising an alarm if a given criticality level is reached) and it assesses the criticality of the network, in terms of the future hazardous scenarios that can happen, considering the new network conditions.

The proposed methodology and tool has been applied on two realistic WSNs representative of health monitoring scenarios. The case studies show how the approach is useful to deeply investigate the reasons of inefficiency and to re-target design choices. They also show how the same specification can be used at runtime to check the correct behavior of a real WSN, deployed on the field and monitored by ADVISES.

The rest of the paper is organized as follows. A discussion of the related work is given in Section 2. Section 3 is dedicated to the description of a process underlying our the proposed methodology. Section 4 reports the definition of the correctness specifications (general and structural) of WSNs following the *event calculus* formalism. Section 5 addresses the static verification in the form of *robustness checking*. The runtime verification technique is discussed in Section 6. Finally in Section 8 the paper concludes with final remarks, discussion of limitations, and indications for future work.

2. Related Work

Several approaches have been proposed in the literature for the dependability evaluation of WSNs properties: experimental, simulative, analytical and formal.

Experimental approaches are used to measure the dependability directly from a real distributed system, during its operation and thus they allow to analyze dependability at runtime [14]. In the field of WSNs, Li and Liu presented in [15] a deployment of 27 Crossbow Mica2 motes that compose a WSN. They describe a Structure-Aware Self-Adaptive WSN system (SASA) designed in order to detect changes of the network due to unexpected collapses and to maintain the WSN integrity. Detection latency, system errors, network bandwidth and packet loss rate were measured; coverage and connection resiliency metrics are not considered. In the prototyping phase, it is possible to perform an accelerated testing, for example by forcing faults in sensor nodes (by means of *Fault Injection* (FI) [16,17]).

Simulative approaches for assessing WSNs usually make use of behavioral simulators, i.e., tools able to reproduce the expected behavior of a system by means of a code-based description, and they are involved in design phase. Typical simulative approaches to evaluate WSN fault/failure models are provided in [18,19]. In [18] authors address the problem of modeling and evaluating the reliability of the communication infrastructure of a WSN. The first on-line model-based testing technique [19] has been conceived to identify the sensors that have the highest probability to be faulty. Behavioral simulators, as NS-2 [20] and Avrora [21], allow to reproduce the expected behavior of WSN nodes on the basis of the real application planned to execute on nodes. However, it is not always possible to observe non-functional properties of WSNs by means of simulative approaches, since models need to be redefined and adapted to the specific network to simulate.

The study of the performance and dependability of WSNs can be performed by means of analytical models [22–24]. In [22] authors introduce an approach for the automated generation of WSN dependability models, based on a variant of Petri nets. An analytical model to predict the battery exhaustion and the lifetime of a WSN, *LEACH*, is discussed in [23]. In [24] the authors present a network state model used to forecast the energy of a sensor.

Formal approaches offer new opportunities for the dependability study of WSNs. Recently, different formal methods and tools have been applied for the modeling and analysis of WSNs, such as [25], [26] and [27]. In [25] Kapitanova and Son apply a formal tool to WSNs. They propose a formal language to specify a generic WSN and a tool to simulate it. However, the formal specification has to be rewritten if the WSN under study changes. In [26] Man et al. propose a methodology for modeling, analysis and development of WSNs using a formal language (PAWSN) and a tool environment (TEPAWSN). They consider only power consumption as dependability metric that is necessary but not sufficient to assess the WSN dependability (e.g. other problems of WSN such as the isolation problem of a node have been analyzed) and also they apply only simulation. In [27] Boonma and Suzuki describe a model-driven performance engineering framework for WSNs (called Moppet). This framework uses the event calculus formalism [28] to estimate, only at design time, the performance of WSN applications in terms of power consumption and lifetime of each sensor node; other dependability metrics like coverage and connection resiliency are not considered. The features related to a particular WSN have to be set in the framework every time that a new experiment starts.

There are some papers ([29],[30]) that considered formal methods in real-time contexts. In [29] Olveczky and Mesenguer model and study WSN algorithms using the Real-Time Maude formalism. Though authors adopt this formalism, they use NS-2 simulator to analyze the considered scenarios making the work very similar to simulative approaches. The work presented in [30] considers a WSN as a Reactive Multi-Agent System consisting of concurrent reactive agents. In this paper dependability metrics are not treated and calculated and authors just describe the structure of a Reactive Decisional Agent by means of a formal language.

An open issue with formal specifications of WSNs is that they need to be adapted when changing the target WSN configuration, e.g., in terms of the number of nodes and topology. To address this problem, it is necessary to provide separated specifications and thus conceive two logical sets of specifications: a general specification for WSN correctness properties that is valid for any WSN, and a structural specification related to the topology of the target WSN, designed in order to be generated automatically. Currently there are proposals in the literature documenting the application of formal methods to model the WSN but they only focus on some dependability metrics as lifetime and power consumption; it is necessary to provide a method of assessing the dependability in terms of other important key dependability metrics, such as coverage and connection resiliency to undesired events. Moreover, no approach has been defined using formal methods for doing static and runtime WSN verification as we propose in this paper.

3. The Process of the Methodology

Formal methods have been widely adopted in the literature to verify the correctness of a system taking into account specifications. The verification is performed by providing a proof on an abstract mathematical model of the system. Until now there is no work that has proven how to use a unique formal approach to perform dependability assessment at design time and at runtime.

This paper defines a new full formal methodology to support verification of WSNs both at design time (through *static verification*) and runtime (through *runtime verification*) exploiting only one set of formal specifications divided in two subsets: *general* (unchangeable and valid for any WSN) and *structural* (variable on the basis of particular configuration of the WSN) specifications.

In this Section we introduce our proposed methodology modeled as a process characterized in steps illustrated in figure 1.

The process is characterized by five sequential phases: *Informal Domain Description*, *Design*, *Static Verification*, *Deployment* and *Runtime Verification*; all the process consists of 13 tasks.

3.1. Informal Domain Description

In the *Informal Domain Description* phase the application is described and its requirements of performance and dependability (e.g. coverage threshold) are defined.

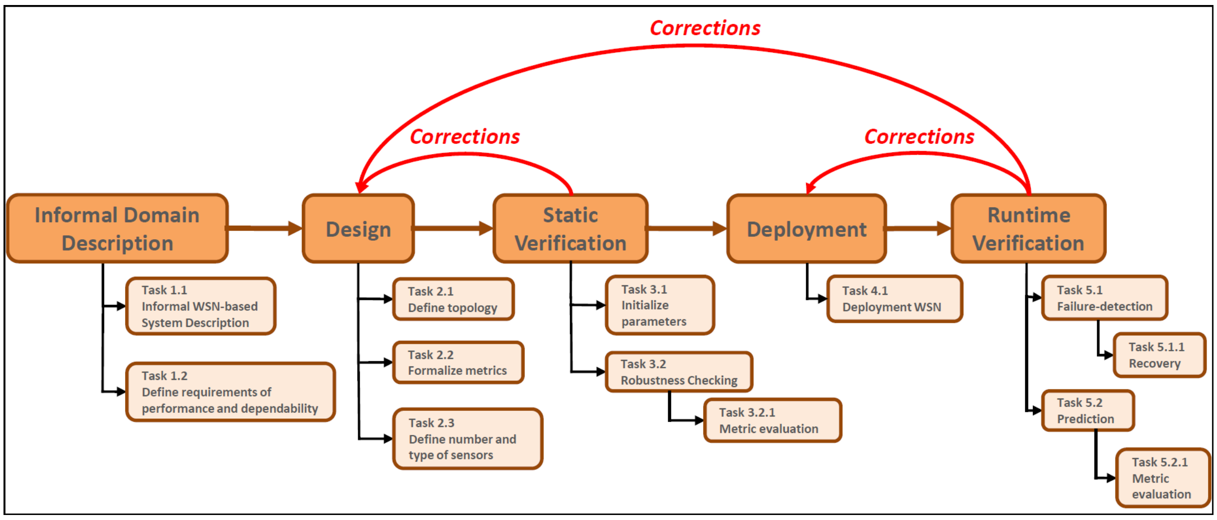

Corrections
Corrections
Corrections
Informal Domain Description
Design
Static Verification
Deployment
Runtime Verification
Task 1.1 Informal WSN-based System Description
Task 1.2 Define requirements of performance and dependability
Task 2.1 Define topology
Task 2.2 Formalize metrics
Task 2.3 Define number and type of sensors
Task 3.1 Initialize parameters
Task 3.2 Robustness Checking
Task 3.2.1 Metric evaluation
Task 4.1 Deployment WSN
Task 5.1 Failure-detection
Task 5.1.1 Recovery
Task 5.2 Prediction
Task 5.2.1 Metric evaluation

Figure 1. The process of the proposed methodology (5 phases - 13 tasks).

3.1.1. Task 1.1: Informal WSN-based System Description

Using natural language the WSN-based system domain is described in a textual form useful to understand what is the considered scenario, what are possible events that can occur, the number of sensors of the network, what they sense, how many redundant nodes there are.

3.1.2. Task 1.2: Define requirements of performance and dependability

After having defined the application domain, in this task the performance and dependability characteristics are described. For example *"We want to guarantee at least 65% of coverage for an area against occurred failure events"*, *"We want that the number of failure events active in same time is not higher than half of the total number of sensors"*.

The dependability metrics considered in this paper, are:

- *connection resiliency* represents the number of node failures and disconnection events that can be sustained while preserving a given number of nodes connected to the sink.
- *coverage* is the time interval in which the WSN can operate, while preserving a given number of nodes connected to the sink.

3.2. Design

This phase includes the design of a WSN defining the topology represented by means of a tree graph with the sink node as root. Since several WSNs are composed by sensors that are in a fixed place (i.e. in case of structural monitoring, hospital environment monitoring, fire monitoring, home environment monitoring, etc), we focus on static routing topologies that can be represented as directed spanning trees [31] (with the sink as root). Dependability metrics are formalize and number and type of sensors are defined.

3.2.1. Task 2.1: Define topology

This task defines the topology; nodes and links are identified. From the structure of the WSN, we retrieve the corresponding spanning tree.

3.2.2. Task 2.2: Formalize metrics

This task formalizes the required metrics on the basis of requirements defined in the previous step (e.g. coverage computation).

3.2.3. Task 2.3: Define number and type of sensors

This task is complementary to the task 2.1 in order to define topology and fix the number of the sensors. In particular, the properties of sensors (e.g. RX/TX energy consumption) are defined since they are considered in the next phase (*Static verification*).

3.3. Static Verification

The designed WSN is verified in terms of the defined dependability properties. The verification is static in the sense that the network is not still operating.

The effects of the reasoning, performed on the basis of the defined specifications (general and structural), impact on the designed WSN. For instance, if the results of the static verification do not meet the dependability requirements, it could be necessary to modify the topology of the WSN, or relax the requirements.

3.3.1. Task 3.1: Initialize parameters

This task focuses on the choice and initialization of the parameters needed to perform static verification. The designed topology is loaded; then users can set several parameters such as numbers of timepoints, sensors and tolerated failures. Moreover dependability thresholds are set.

3.3.2. Task 3.2: Robustness Checking

In this task, the WSN design is verified by means of an event-based formal approach. Dependability metrics (formalized in the task 3.2.2) are evaluated against random sequences of undesired events, useful to identify corner cases and dependability bottlenecks.

3.3.3. Task 3.2.1: Metric evaluation

This task performs the computation of the dependability metrics (such as coverage and connection resiliency) analyzing the outcome produced by the robustness checking process. The metrics are calculated on the basis of their definitions and considering the output produced by the formal reasoner. Once the metric values have been obtained, this task provides the comparison among them and the threshold values set by the user.

3.4. Deployment

In this phase, the WSN, verified at design time, is actually deployed distributing wireless sensors in one or more environments.

3.4.1. Task 4.1: Deployment WSN

The aim of this task is to physically deploy the WSN in one or more environments.

3.5. Runtime Verification

The last phase concludes the process. The aim is to formally verify at runtime the real WSN hopefully with results which are consistent to those achieved at design time. Like in the *Static Verification* phase, the effects of the reasoning on the defined specifications (general and structural) may impact the implemented deployment of the WSN: if unexpected results occur, then further changes may be performed at the deployment or even at the design stage. If it is necessary, the topology will have to be changed and thus both statically and dynamically validate it again on the basis of defined requirements.

3.5.1. Task 5.1: Failure-detection

In this task, failure events (node failures, disconnections, ...), occurring in the wireless sensors, are detected during the system running and it is generated an event in a particular formalism in order to start the computing of the new dependability degree.

3.5.2. Task 5.1.1: Recovery

In this task, if the current value of metrics is lower than the desired threshold, the network characteristics (topology, position of the nodes, power of transmission, etc.) can be modified to let the WSN be able to satisfy the required dependability level.

3.5.3. Task 5.2: Prediction

This task operates like task 3.2. The robustness checking is performed in order to predict the dependability level in case of future failures in the WSN.

3.5.4. Task 5.2.1: Metric evaluation

This task operates like task 3.2.1.

4. Specifications

In this section we describe the formal specification of WSN correctness composed by two logical sets: in the first one, we define *invariant* rules that are applicable to any WSN and thus written only one time; in the second set, we define *variable* specifications that are dependent on a given WSN structure (i.e. topology, number of nodes, sent packets, etc.). All the defined formal specification underlies verification process described in the previous section to perform static and runtime verification:

1. ***general correctness specification*** - a set of correctness properties specifications, valid independently of the particular WSN under study
2. ***structural specification*** - a set of specifications and parameters related to the properties of the target WSN, e.g., number of nodes, network topology, quality of the wireless channel (in terms of disconnection probability), and initial charge of batteries. It has to be adapted when changing the target WSN having thus the advantage of maximizing the effort.

4.1. Event Calculus

Since the normal and failing behavior of a WSN can be characterized in terms of an event flow (for instance, a node is turned on, a packet is sent, a packet is lost, a node stops to work due to crash or battery exhaustion, or it gets isolated from the rest of the network due to the failure of other nodes, etc.), we adopt an *event-based* formal language. In particular, among several event-based formal languages, we choose *event calculus*, since its simplicity, its wide adoption in the sensor networks arena ([27], [32], [33] and [34]), and the possibility to formally analyze the behavior of a system as event flows, offering simple ways to evaluate the dependability metrics of our interest even at runtime.

Event calculus was proposed for the first time in [35] and then it was extended in several ways [36]. This language belongs to the family of logical languages and it is commonly used for representing and reasoning about the events and their effects [37].

Fluent, *event* and *predicate* are the basic concepts of event calculus [28]. Fluents are formalized as functions and they represent a stable status of the system. For every timepoint, the value of fluents or the events that occur can be specified.

This language is also named *narrative-based*: in the event calculus, there is a single time line on which events occur and this event sequence represents the *narrative*. Dependability metrics can be valuated by analyzing the *narrative* generated by an event calculus reasoner based on the specification of the target WSN. A narrative is useful to understand a particular behavior of a WSN.

The most important and used predicates of event calculus are: *Initiates*, *Terminates*, *HoldsAt* and *Happens*.
Supposing that e is an event, f is a fluent and t is a timepoint, we have:

- *Initiates(e, f, t)*: it means that, if the event e is executed at time t, then the fluent f will be true after t.
- *Terminates(e, f, t)*: it has a similar meaning, with the only difference being that when the event e is executed at time t, then the fluent f will be false after t.
- *HoldsAt(f, t)*: it is used to tell which fluents hold at a given time point.
- *Happens(e, t)*: it is used when the event e occurs at timepoint t.

Several techniques are considered to perform automated reasoning in event calculus, such as satisfiability solving, first-order logic automated theorem proving, Answer Set Programming and logic programming in Prolog.

To check the proposed correctness properties defined in event calculus we use the *Discrete Event Calculus (DEC) Reasoner* [38]. The DEC Reasoner uses satisfiability (SAT) solvers [39] and by means of this we are able to perform reasoning like deduction, abduction, post-diction, and model finding. The DEC Reasoner is documented in detail in [40] and [41] in which its syntax is explained (e.g. the meaning of the symbols used in the formulas).

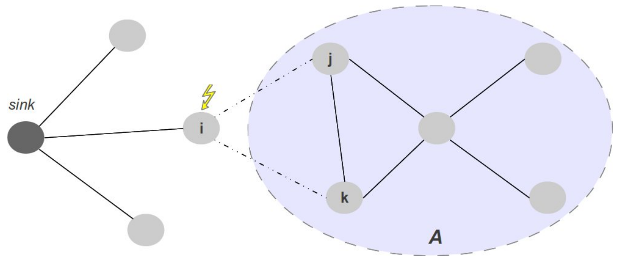


Figure 2. Isolation of a WSN subnet

4.2. General Correctness Specification

The general correctness specification is described in the following. It specifies that a WSN performs correctly if none of the following main undesired events (or *failures*) happens:

1. *isolation event*, i.e., a node is no more able to reach the sink;
2. *packet loss event*, i.e., a packet is lost during the traversal of the network;
3. *battery exhaustion event*, i.e., a node stops to work since it has run out of battery.

The first two kinds of event can be caused by more "basic events", such as the stop of one or more nodes (e.g., due to crash or battery exhaustion) or the temporary disconnection of a node to its neighbor(s) due to transmission errors. The last kind of event is dependent on the power consumption of the nodes as a consequence of packet sending and receiving activities (in general assumed to be power demanding activities with respect to CPU activities [42]).

For matter of space, in this paper we concentrate on first of the three main events described previously (*isolation*) considering the results of a *Failure Modes and Effect Analysis* conducted on WSNs in [43]; specifications of the other two events (*packet loss* and *battery exhaustion*), defined also in [43], are omitted.

The isolation event happens when a node is no more able to reach the *sink* of the WSN, i.e., the gateway node where data are stored or processed. The isolation might be caused by more simple, basic events, such as a stop of a node, due to an arbitrary crash or battery exhaustion, and the disconnection of a node from another node.

For instance, let us consider the figure 2 and let us suppose that node i is the only one allowing the transmission of data between the sink node and the subnet A. We want to check when the subnet A is isolated from the rest of the network. Let us suppose that node i is connected with node j and k. If node i fails, the nodes j and k (and all the nodes of the subnet A) are alive but isolated and so the whole subnet A is isolated.

More in general, if a subnet depends on a node and this node becomes isolated then all of the nodes of the subnet are isolated.

In table 1 we report the basic elements (sorts, events and fluents) used for the specification of situations like these. We distinguish basic events from generated events. These last events are generated by the reasoner on the basis of the specification and of the sequence of basic events which actually occurred.

Listing 1 shows the rules that represent the core of the specification for an isolation event. In lines 1-7 we define a rule to verify when a node becomes isolated. A sensor can be isolated if it is initially reachable, alive and, considering a link with another sensor, there is no other sensor that is alive, reachable and connected with the isolated sensor.

Also we report (in lines 9-14) another rule which allows to check a *Join* event. In particular a sensor, that is isolated, can rejoin the network and become reachable again if it still is alive, its neighbor sensor is alive and reachable from the sink node and a connection between them has been restored.

Table 1. Basic elements of the specification for the isolation event

Elements	Name	Description
Sorts	*sensor*	Reference sensor for events and fluents
	to_sensor	Sensor used in case of connection (i.e. a sensor connects **to** another sensor)
	from_sensor	Sensor used in case of disconnection (i.e. a sensor disconnects **from** another sensor)
Basic Events	*Start(sensor)*	Occurring event when a sensor turns on
	Stop(sensor)	Occurring event when a sensor turns off
	Connect(sensor, to_sensor)	Occurring event when a sensor connects to another sensor
	Disconnect(sensor, from_sensor)	Occurring event when a sensor disconnects from another sensor
Generated Events	*Isolate(sensor)*	Occurring event when a sensor is isolated from the network
Fluents	*IsAlive(sensor)*	True when a *Start* event occurs for a sensor
	IsLinked(sensor, to_sensor)	True when a *Connect* event occurs
	IsReachable(sensor)	True when a sensor is reachable from the sink node

In lines 16-28 we show conditions in which *Isolation* and *Join* events cannot occur. We claim that an *Isolation* event cannot occur in a sensor (in lines 16-21) if at least one of its neighbor sensors is alive and reachable and is connected with the sensor. Moreover, if a sensor is not reachable, due to a previous isolation, or not alive, it cannot receive another *Isolation* event again. In the similar way (in lines 23-28), if a sensor is reachable or not alive or all of its neighbor sensors are not alive or not reachable, then it cannot join to the network remaining isolated.

Listing 1: Correctness Specification for the Isolation event

```
[sensor,from_sensor, time] Neighbor(from_sensor,sensor) & HoldsAt(
        IsReachable(sensor),time) & HoldsAt(IsAlive(sensor),time) &
        (!{from_sensor2} (HoldsAt(IsAlive(from_sensor2),time) &
        HoldsAt(IsReachable(from_sensor2),time) & HoldsAt(
        IsLinked(sensor,from_sensor2),time)) &
        Neighbor(from_sensor2,sensor)) ->
Happens(Isolate(sensor),time).

[sensor,from_sensor, time] ( !HoldsAt(IsReachable(sensor),time) &
        HoldsAt(IsAlive(sensor),time) & HoldsAt(IsAlive(
        from_sensor),time) & HoldsAt(IsReachable(from_sensor),time))
         & HoldsAt(IsLinked(sensor,from_sensor),time) &
        Neighbor(from_sensor,sensor) ->
Happens(Join(sensor),time).

[sensor,from_sensor, time] ((HoldsAt(IsAlive(from_sensor),time) &
        HoldsAt(IsReachable(from_sensor),time) & HoldsAt(
        IsLinked(sensor,from_sensor),time)) | !HoldsAt(
        IsReachable(sensor),time) | !HoldsAt(IsAlive(sensor),time))
         & Neighbor(from_sensor,sensor) ->
!Happens(Isolate(sensor),time).
```

Listing 2: Use of *Neighbor* predicate in a structural specification

```
[sensor1,sensor2] Neighbor(sensor1,sensor2) <-> (
(sensor1=i & sensor2=j) |
(sensor1=i & sensor2=k)
).
```

```
[sensor,from_sensor,time] ( HoldsAt(IsReachable(sensor),time) |
         !HoldsAt(IsAlive(sensor),time) | !HoldsAt(
         IsLinked(sensor,from_sensor),time) |
         !HoldsAt(IsAlive(from_sensor),time) | !HoldsAt(
         IsReachable(from_sensor),time)) & Neighbor(from_sensor,sensor)->
!Happens(Join(sensor),time).
```

4.3. Structural Specification

General correctness specifications are complemented by a structural specification that comprises a set of specifications and parameters related to the properties of the target WSN, e.g., number of nodes, network topology, quality of the wireless channel (in terms of disconnection probability), and initial charge of batteries. This specification depends on particular WSN topology and thus, differently from the specifications described in the previous sub-section, it varies on the basis of the characteristics of the target WSN.

To specify the topology, we use the predicate *Neighbor* (already used in the previous specifications) to indicate how nodes are linked in the topology. For instance, considering the topology in figure 3: node *i* is connected with *j* and *k* and the sink (root node) is the node *i*.

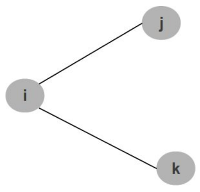


Figure 3. Example of topology of a WSN

The resulting specification is reported in listing 2, where *sensor1* is the parent node (*i*) and *sensor2* are child nodes (*j* and *k*). Clearly, this specification can be changed easily if the topology of the WSN changes.

The role of the *Neighbor* predicate is very important to understand when an axiom can be applied. Let us examine the axiom related at a possible isolation (lines 1-7 of listing reported in Listing 1) and let us apply it for the figure 3. The described implication is true when, given a couple of nodes (*sensor*, *from_sensor*), the conditions about isolation are true and there is a link between nodes (in this case, between node *j* and *i* or between node *k* and *i*). This, for instance, can never be true for the couple of nodes *j* and *k*, since there is not a physical link between them.

Regarding the parameters, their values can be used to check the correctness properties of the WSN under different conditions, i.e., under different assumptions on the initial charge of batteries (e.g., to verify a WSN in the middle of its life), or under different environmental conditions affecting the quality of the channels (impacting on the probability to have a disconnection event when checking the robustness of the WSN).

Listing 3: Structural specification of the WBSN topology

```
[sensor1,sensor2] Neighbor(sensor1,sensor2) <-> (
(sensor1=1 & sensor2=2) |
(sensor1=2 & sensor2=3) | (sensor1=2 & sensor2=4) |
(sensor1=3 & sensor2=5) |
(sensor1=4 & sensor2=6) |
(sensor1=6 & sensor2=7)
).
```

4.4. Example: A Wireless Body Sensor Network

Let us consider a simple example to show the use of the specification on a WSN and how the narrative produced by the event calculus reasoner is useful to compute the metrics of interest. In particular, we consider a wireless body sensor network (WBSN) realized by Quwaider et al. [44] and illustrated in figure 4.

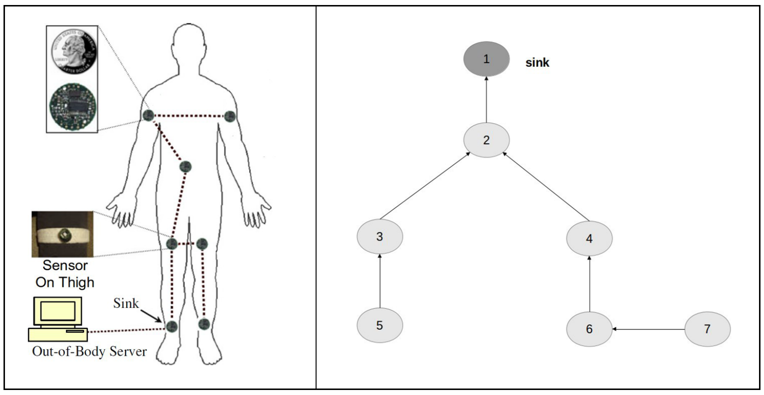


Figure 4. Quwaider WBSN and related topology

This WBSN (figure 4 on left side) is constructed by mounting seven sensor nodes attached on two ankles, two thighs, two upper-arms and one in the waist area. Each node consists of a 900 MHz Mica2Dot MOTE (running Tiny-OS operating system).

On right side of the figure 4 we report the node tree graph corresponding to the WSN, where the arrows indicate the relationship between a couple of nodes (i.e. node 2 depends by node 1, node 3 and 4 depend by node 2, etc.).

The corresponding specification of the topology is reported in the listing 3.

Assuming a coverage threshold of 50%, let us suppose to be interested at analyzing the behavior of the WBSN if the following events occur: *Disconnect(5,3)* at timepoint 1 and *Stop(4)* at timepoint 3.

If the specification is correct, we should observe a coverage interval equals to $[0,3]$ (i.e., when node 4 stops at time point 3, 4 nodes are not reachable, namely 4, 5, 6 and 7), and a connection resiliency equals to 1 (i.e., only on event - the *Disconnect(5,3)* event - is tolerated). To test the desired event sequence, we add an event trace (listing 4) to the specification, composed by a list of *Happens* predicates that specify nodes and timepoints in which a given event occurs. The *completion* statement specifies that a predicate symbol (i.e. *Happens*) should be subject to predicate completion.

Listing 4: Event trace

```
Happens(Disconnect(5 , 3),1).
Happens(Stop(4),3).

completion Happens
```

Listing 6: Outcome of the DECReasoner

```
0
1
Happens(Disconnect(5, 3), 1).
2
-IsLinked(5, 3).
Happens(Isolate(5), 2).
3
-IsReachable(5).
Happens(Stop(4), 3).
4
-IsAlive(4).
Happens(Isolate(6), 4).
5
-IsReachable(6).
Happens(Isolate(7), 5).
6
-IsReachable(7).
7
8
9
10
```

Finally, in the last part we consider ranges of values for sensors and timepoints (listing 5). In this case we know that the network is composed by 7 nodes and we want to observe what it could happen in 10 timepoints.

Listing 5: Parameters

```
range sensor 1 7
range time 0 10
```

Listing 6 reports the outcome (the *narrative*) produced by the DECReasoner. The event trace confirms our expectations. We can observe that after the stop of node 4, nodes 6 and 7 become not reachable. Considering that node 5 was already not reachable, this means that a total of 4 nodes are isolated. The coverage is computed as the time point of the last failure event causing such isolation, that is 3. Consequently, the connection resiliency is computed by counting the number of failure and disconnection events in the interval $[0, 3]$, excluding the last event; hence, it is equal to 1, as expected.

5. Static Verification

In the previous example we have shown how the specifications and the reasoning performed on them can be exploited to analyze the WSN response to a given sequence of undesired events. This concept can be extended to test the WSN against a variable sequence of events, in order to verify its design (*static verification*) in the form of a *coverage robustness checking*.

Specifically, the verification consists in analyzing the robustness of the network, in terms of coverage, against a variable number of failures (stop and disconnection events), from 1 to n, where n is selected by the user, considering all combinations without repetitions. This is useful to check how many node failures the network can tolerate, while guaranteeing a given minimum level of coverage. For example if we consider a network composed by m nodes and a threshold coverage equal to 50%, we may want to understand what are the sequences of failures causing more than $m/2$ nodes to be isolated (i.e., coverage under the specified threshold) and how the resiliency level varies when varying the sequences of failures. This allows to evaluate the maximum (and minimum) resiliency level reachable by a given topology and what are the critical failure sequences, i.e., the shortest ones causing a loss of coverage. These are particularly useful to pinpoint weak points in the network (so called *dependability bottlenecks*).

We developed an algorithm to generate automatically the sequences of failures (stop and disconnection events specified with *Happens* predicates) against which checking the robustness of the WSN. The algorithm is implemented by the ADVISES tool (see section 7.1), and it is aimed to reduce the number of failure sequences to be checked. The principles are to avoid repetitions and to end the sequence as soon as the coverage level becomes lower than the user defined threshold. For instance, we start considering all the cases when there is one failure. By means of the DECReasoner we compute the coverage; if the coverage is above the threshold, the resiliency is surely greater than 1, because there is just one failure and it is tolerated in all cases. In the generic $k - th$ step, we consider sequences of k failures. If the generic sequence $\{f_1, f_2, ..., f_k\}$ leads to a coverage below the threshold, we do not consider sequences starting with an $\{f_1, f_2, ..., f_k\}$ prefix in the $(k + 1) - th$ step. By considering the percentage of sequences with k failures where the coverage is above the threshold, let us say r_k%, we can say that the resiliency is k in r_k% of cases.

6. Runtime Verification

The aim of this step is i) to perform a *Runtime Verification* (RV) [11,45] detecting failure events occurring in a real WSN, possibly to activate recovery actions and ii) to perform a prediction of the critical levels of next failure events that may occur in the WSN, to take countermeasures in advance. In this case, critical events, e.g., *Stop* and *Disconnect*, are not simulated anymore with *Happens* predicates, but they are detected from the real system, through *system monitors* (such as the ones we have defined in [46]).

An application scenario is considered to show how the runtime verification can be implemented; the aim is to describe how it is possible to catch events and observe their effects in a WSN at runtime.

We can see, in figure 5 from the left to right, that when an event occurs in a wireless sensor node it is detected by a system monitor that runs on a gateway. The failure event (for instance *Stop(n)*) is managed by the monitor and added to the current event trace to perform the reasoning. The new event trace is included in an updated structural specification; thus the DEC Reasoner receives the structural specification with the last occurred event and considering the general correctness specifications (initially defined) performs the reasoning returning a couple of outcomes: i) the *Current Dependability Level* of the WSN and ii) the *Potential Critical Nodes*.

The first outcome reports the current WSN dependability level (i.e. the WSN now covers the X% of the monitored area, and it has been resilient to Y failures so far). The second outcome is a prediction about possible critical events that may occur after the current event (e.g., from now on, node Z represents a weak point in the WSN: it should be replicated or its batteries should be replaced). Moreover the runtime verification is useful to further verify at runtime the WSN design that has been validated at design time. Even if a WSN is checked at design time, it is necessary to observe if the implemented WSN conforms to expectations and to continuously monitor if it is able to cope with unexpected events.

7. Case studies

In this section, by means of two case studies, we apply the methodology described in Section 3 to study and improve the robustness of two WSNs.

We have performed our experiments on a Intel P4 machine, CPU Clock 3.5 GHz, 512 MB RAM, equipped with Linux, kernel 3.8.8.

We have focused on WSN-based healthcare systems due to their criticality related to patient monitoring scenarios.

In table 2 we report a list of studied papers focused on WSN-based healthcare systems. After a general research we have chosen the following systems: the *Self-powered WSN* for remote patient

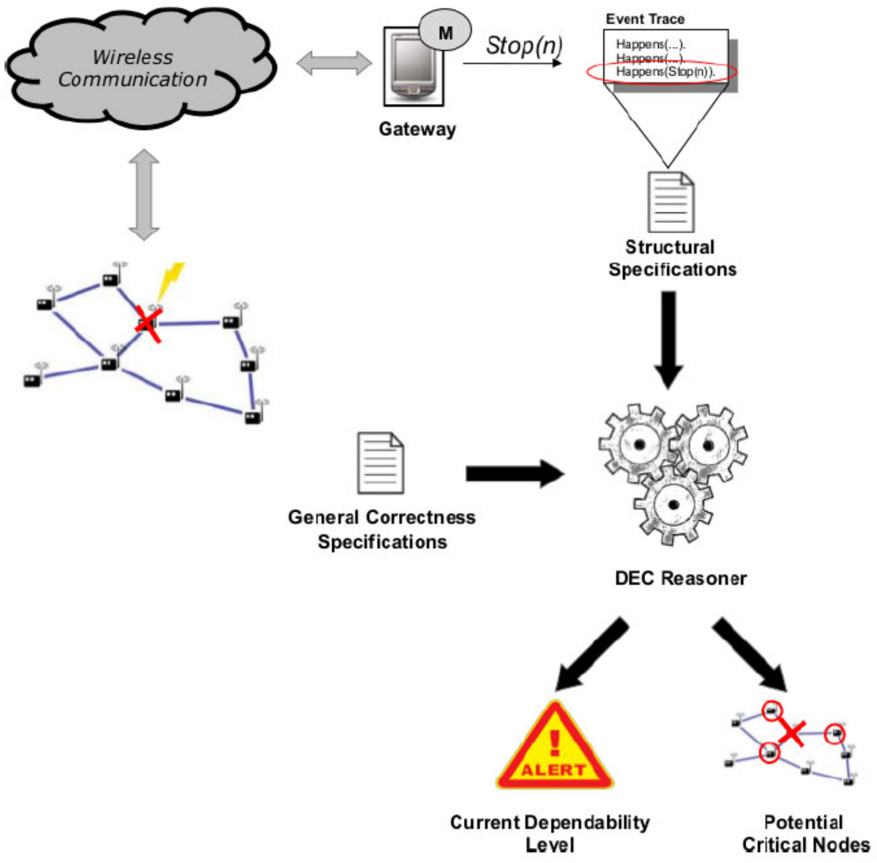


Figure 5. Application scenario in runtime context

Table 2. Analysis of some WSN healthcare systems

Work	Nodes	Topology	Sensor Platform
iNODE-based system [47]	4	tree	iNODE
BSN-based system [48]	8	fully-connected	Jennic JN5139
MEDiSN [49]	10	tree	Sentilla Tmote Mini
HM4ALL [50]	12	tree	JN5139-MOI ZigBee-based platform
Self-powered WSN [51]	13	tree	Crossbow Micaz
Multi-patient system [52]	15	grid	Tmote sky
CodeBlue [53]	16	grid	N.A.
Clinical Monitoring System [54]	18	tree	TelosB mote

monitoring (adopted in hospitals)[51] and the *MEDiSN*[49] (a WSN for automating process of patient monitoring in hospitals and disaster scenes).

In order to facilitate and automatize the application of the proposed methodology (comprising static and runtime verification) against a general case study, a Java-based tool, called *ADVISES*[2] (**A**utomate**DV**er**I**fication of w**S**n with **E**vent calculu**S**), has been designed and implemented.

[2] http://sourceforge.net/projects/advises

7.1. The ADVISES Tool

The goal of the ADVISES tool [43] is to provide technical support to the methodology addressing practical aspects (e.g. setting of parameters necessary to start verification, realization of structural specification in automatic way and merge with general specification).

This tool has been realized i) to operate in double mode: static and runtime; ii) to automatically generate the structural specifications given the properties of a target WSN; iii) to perform the reasoning starting from the correctness and structural specifications; iv) to compute dependability metrics starting from the event trace produced by the reasoner; and v) to receive events in real-time from a WSN to start runtime verification and to evaluate current and future criticalities.

In particular, at runtime, it is like a server that is in waiting for new events coming from the WSN and that are detected by means of a system monitor.

Since the static and runtime verification do not require the same number of input parameters, we present the ADVISES tool operating in double mode: in static mode we need to select several parameters that in runtime mode are not necessary.

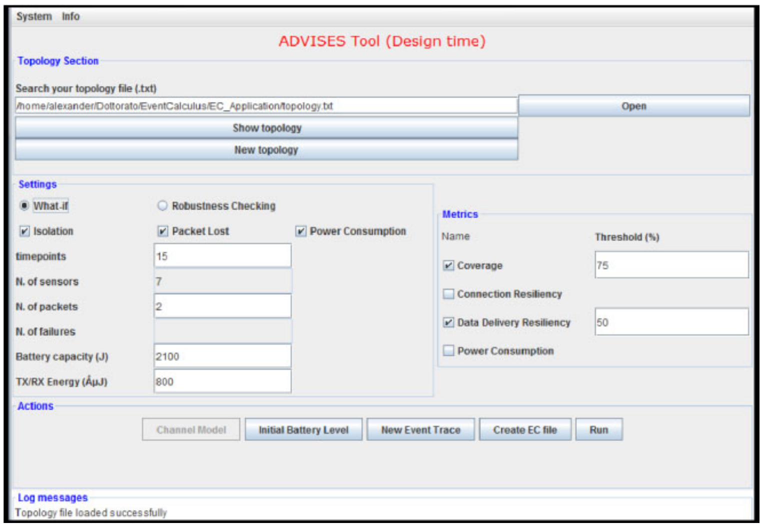
7.1.1. ADVISES for Static Verification

The ADVISES tool in static verification needs several parameters like the number of packets, the number of failures (for performing the robustness checking at design time), the battery capacity value of a node, the initial event trace, etc.

By means of the interface shown in figure 6, a user can simply specify i) the topology of the target WSN (using a connectivity matrix), ii) the formal correctness specifications (e.g. for checking isolation events), iii) the temporal window size to consider (in terms of the number of timepoints), iv) the number of packets that each sensor can send, v) the number of failures (in case of coverage robustness checking), vi) the battery capacity of a sensor (in Joule) and the needed energy for RX/TX operations (in μJoule), vii) the metrics to calculate (for coverage it is necessary also the threshold value), viii) the channel model (in case of robustness checking), ix) the initial battery level, x) the initial event trace (to perform what-if scenario analysis).

The interface is subdivided by five panels:

- *Topology* section (the user selects or creates a new topology matrix);
- *Settings* section (by means of this the user can set the several parameters that will be considered for the static verification);



Search your topology file (.txt): /home/alexander/Dottorato/EventCalculus/EC_Application/topology.txt

[✓ What-if] Robustness Checking

[✓ Isolation] [✓ Packet Lost] [✓ Power Consumption]

timepoints	15
N. of sensors	7
N. of packets	2
N. of failures	
Battery capacity (J)	2100
TX/RX Energy (ÅµJ)	800

Name	Threshold (%)
[✓ Coverage]	75
Connection Resiliency	
[✓ Data Delivery Resiliency]	50
Power Consumption	

Topology file loaded successfully

Figure 6. ADVISES in static mode

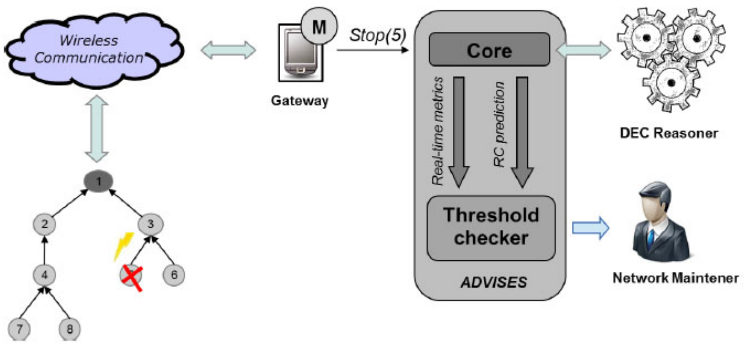


Figure 7. Use of the ADVISES Tool at runtime

- *Metrics* section (the user chooses with a tick in a box the desired metrics and in particular in case of coverage he has to write the percentage values (default value is *50*));
- *Actions* section (the ADVISES tool presents the possible steps that user can perform);
- *Log messages* panel, the ADVISES tool reports all the useful messages in order to inform the user if some error occurs.

7.1.2. ADVISES for Runtime Verification

The ADVISES tool in runtime verification mode is simpler because it works on the basis of the events that receives from the WSN.

Figure 7 shows the ADVISES operating at runtime.

Once started, it is in *server mode* and waiting for events coming from the WSN; when a failure occurs in a wireless sensor node, (for instance *Stop(5)*), it is managed by the monitor and sent to the ADVISES Tool that is listening on a port ready to receive events and start reasoning. Received the event, it automatically updates the sequence of received events and generates the event calculus specification file in order to perform the reasoning with the DEC Reasoner. The ADVISES tool, received the output from the DEC Reasoner, calculates the current values of the selected dependability metrics to establish the current status of the WSN and performs robustness checking to predict the future criticalities. Then, verified the obtained metrics values with the thresholds set by the user, the ADVISES tool sends messages to a network maintainer which may be purely informative or alerts in case these values are under the desired threshold. After the last step, the ADVISES tool continues to work waiting next detected events.

Therefore the ADVISES tool *"advises"* the network maintainer of problems related to the network and reports its critical points.

For this purpose we have realized another interface of the ADVISES tool in order to receive events from a WSN in real-time detected through a system monitor and to start the runtime verification evaluating both current and future criticalities.

Figure 8 shows this interface.

By means of this interface, a user can simply specify i) the initial topology of the target WSN, using a connectivity matrix, ii) the temporal window size to consider, in terms of the number of timepoints to consider for the prediction, iii) the metrics to calculate.

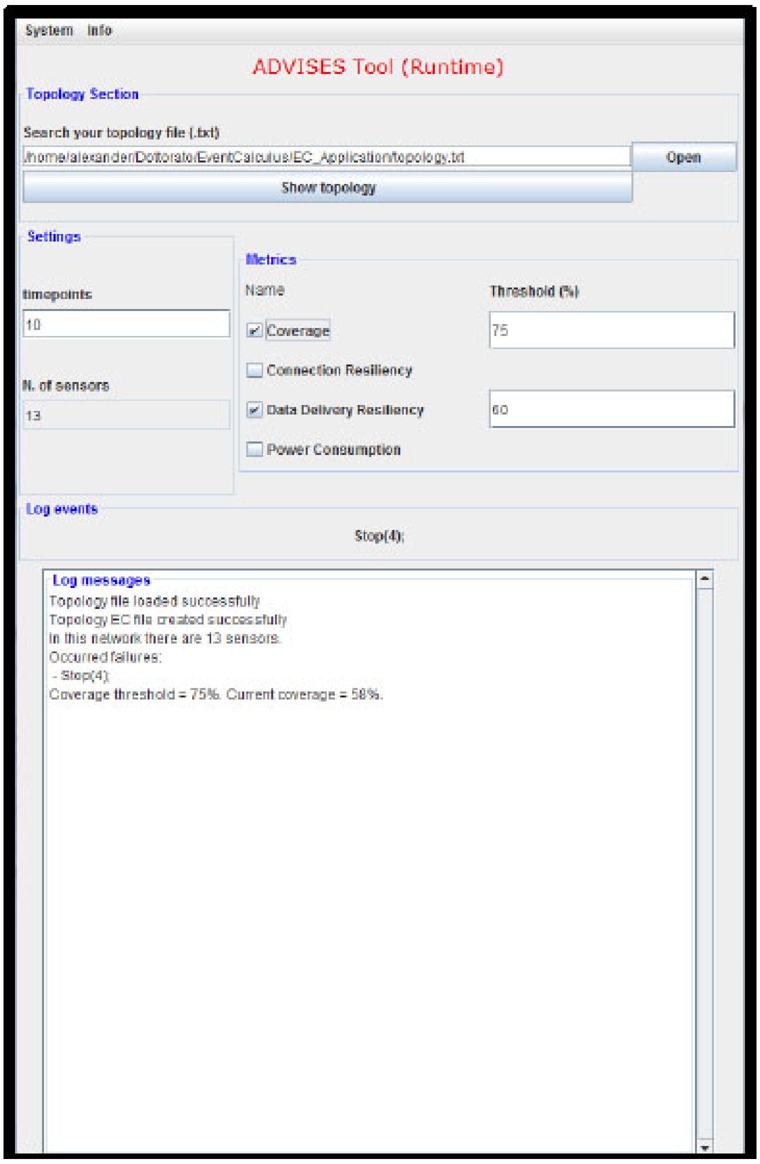
In the *Log messages* panel, the ADVISES tool reports all the performed computations about current state of the WSN and the risks that may affect its robustness. Also there are useful messages in order to inform the user if some error occurs.

7.1.3. Metrics computation

In order to comprise all of the capabilities of the ADVISES tool, it is interesting to know how it is able to calculate the defined dependability metrics.

By means of a parser that analyzes the traces produced by the DEC Reasoner, the ADVISES tool calculates the *coverage* and *connection resiliency*. The computation of these metrics depends on a threshold parameter, to be indicated as a percentage chosen by the user. The threshold expresses the fraction of failed and isolated nodes that the user can tolerate, given its design constraints. For instance, over a WSN of 20 nodes, a threshold set to 100% means that all the 20 nodes have to be connected, whereas 50% means that the user can tolerate at most 10 isolated nodes.

Considering the threshold value, we calculate the coverage analyzing the *IsReachable(sensor)* and *IsAlive(sensor)* fluents found to be true in the event trace produced by the reasoner: if a *-IsReachable(x)* or a *-IsAlive(sensor)* fluent is true in the event trace, this means that node x became isolated or it stopped. For example in the case of coverage at 50%, for a WSN with 7 nodes, there is coverage when at least 4 nodes are not isolated (i.e., they are reachable). Hence, as soon as 4 different nodes are no reachable nor alive (looking at the fluents), the network is not covered anymore. The coverage can be then evaluated as the interval $[0, t]$, being t the timepoint of the last failure or disconnection event before the isolation (e.g., the timepoint of the event that caused the isolation of a number of nodes exceeding the threshold).



Search your topology file (.txt)
/home/alexander/Dottorato/EventCalculus/EC_Application/topology.txt

timepoints: 10

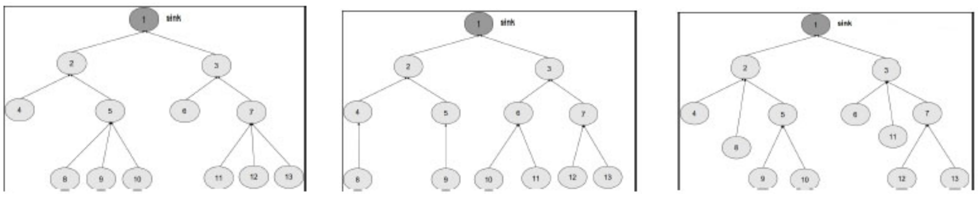
N. of sensors: 13

Name	Threshold (%)
✓ Coverage	75
Connection Resiliency	
✓ Data Delivery Resiliency	60
Power Consumption	

Stop(4);

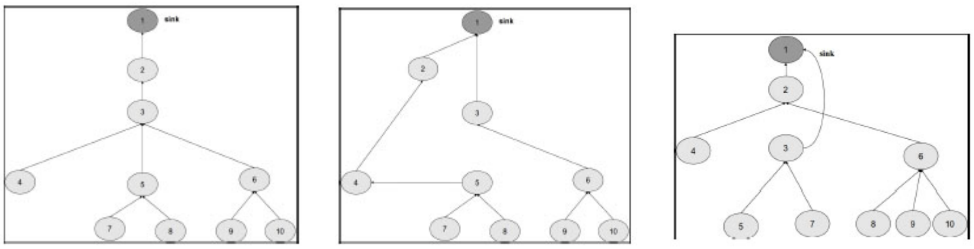
```
Topology file loaded successfully
Topology EC file created successfully
In this network there are 13 sensors.
Occurred failures:
 - Stop(4);
Coverage threshold = 75%. Current coverage = 58%.
```

Figure 8. ADVISES in runtime mode



(a) Original **(b)** First attempt **(c)** Second attempt

Figure 9. Topologies of the self-powered WSN



(a) Original **(b)** First attempt **(c)** Second attempt

Figure 10. Topologies of MEDiSN

The connection resiliency can then be evaluated as the number of failure and disconnection events (namely, *Stop(sensor)* and *Disconnect(sensor, from_sensor)* events) that happen within the coverage interval, excluding the last failure/disconnection event, that is, the one that actually leads the number of isolated nodes to overcome the threshold. For example, if we have coverage in the interval [0, 6], and during this period 3 failure/disconnection events can be counted, then the connection resiliency is 2, that is, the WSN was able to tolerate 2 failures or disconnections while preserving more than 50% of the nodes connected.

7.2. Experimental Results

We perform two sets of experiments. For the first one we consider the self-powered WSN[51] that is composed by 13 motes: 1 base station (that consists of a mote and is connected to a server), 4 router nodes (realized utilizing CrossBow MICAz motes) and 8 sensor nodes (such as ECGs, pulse-oximeters, etc.). For the second one we consider MEDiSN[49] that is composed by 10 motes: 1 base station (as gateway), 4 relay points (that are wireless sensors) and finally 5 physiological monitors (collecting patient's physiological data). For this network, we have considered a topology in which physiological monitors in the leaves are treated as if they were relay points as well so that the topology is more complex and interesting to try our system with.

Table 3. Percentages of connection resiliency for both networks

Outcome		Self-powered WSN			MEDiSN		
		Topology 1	**Topology 2**	**Topology 3**	**Topology 1**	**Topology 2**	**Topology 3**
Cov. = 65%	**Conn. Resil. = 1**	83%	83%	83%	77%	66%	77%
	Conn. Resil. = 2	52%	61%	67%	41%	30%	49%
	Conn. Resil. = 3	31%	35%	41%	18%	10%	28%
Cov. = 75%	**Conn. Resil. = 1**	66%	83%	83%	55%	44%	66%
	Conn. Resil. = 2	43%	48%	50%	29%	18%	43%
	Conn. Resil. = 3	27%	18%	29%	0%	0%	0%
Cov. = 85%	**Conn. Resil. = 1**	66%	66%	66%	55%	44%	66%
	Conn. Resil. = 2	43%	27%	43%	0%	0%	0%
	Conn. Resil. = 3	0%	0%	0%	0%	0%	0%
Examined failure sequences		**114 480**			**46 980**		

7.2.1. Static Verification

For both networks, we aim to observe the percentage of cases in which there is connection resiliency to 1, 2 and 3 failures, keeping a coverage threshold at 65%, 75%, and 85%, so from a less demanding to a more demanding resiliency requirement. Moreover, starting from the original topologies of the two networks as presented in [51] and [49], we attempt to make them more robust, reconfiguring the connections among the nodes and observing effects.

We design the topologies, related to the two chosen networks, by means of a tree graph. The first topology is structured as a tree graph with 3 levels (figures 9a); the second topology (figure 10a) is structured as a tree graph with 4 levels.

Exploiting the capabilities of the ADVISES tool, we specify the characteristics of both networks and of the metrics to be evaluated, specifically connection resiliency with different coverage thresholds. In table 3 we present results of the robustness checking: on the columns we identify the topologies (original and our proposed alternatives) grouped by the network (Self-powered WSN and MEDiSN); on the rows, we collect the results of connection resiliency for 1, 2 and 3 failures on the basis of coverage threshold (65%, 75% and 85%); the generic cell of the table represents the robustness of the network, evaluated as the percentage of cases in which the topology (identified by the column) is able to tolerate *n* failures (where *n* corresponds at the value of the connection resiliency identified by the row) guaranteeing a certain coverage value (identified by the row).

Results on column *Topology 1* are achieved performing robustness checking and dependability evaluation of original topologies. We can observe that, if we increase the requirement on connection resiliency (from 1 to 3) and on coverage (from 65% to 85%), the percentage of cases in which the WSN is able to tolerate the failures decreases, as expected. We can also note that the Self-powered WSN is more robust of the MEDiSN one, due to the higher number of nodes and the lower number of levels of the three graph.

Starting from these results, we try to improve the robustness of both networks, by means of slight changes on the network topology. The aim is to show how the proposed approach, and related tool, are useful to drive design choices and tune the configuration of the network. Considering the Self-powered WSN, we start by proposing a more uniform distribution of the nodes, balancing the tree. The resulting topology is shown in figure 9b. What we expect to observe is a general increase of robustness, since a failure of a node causes the disconnection of a smaller number of nodes with respect to the original topology (e.g. see nodes 5 and 7). For coverage threshold at 65% we achieve this conclusion (see column *Topology 2* in table 3). However, opposite to expectation, for coverage threshold at 75% (and 3 failures) and coverage threshold at 85% (and 2 failures), we obtain a degradation of robustness, from 27% to 18% and from 43% to 27%, respectively.

The automatic analysis performed by the reasoner on the specifications allow us to deeply investigate the exact reason of such degradation, and find better solutions. To this aim, in table 4 we report a classification of the failure events tested by the ADVISES tool on the Self-powered WSN. The events are grouped based on the obtained coverage value (reported in the *Coverage* column) for each of the topologies considered, and are also grouped as below or above the 75% coverage threshold. Passing from topology 1 to topology 2, we can observe that topology 2 is able to reduce the overall number of events causing a coverage value below the threshold (*critical events*), if compared to topology 1, improving the resiliency in several cases. However, we also note that topology 2 introduces a new criticality, not present in topology 1, related to node 3: if this node fails or gets disconnected, the coverage suddenly drops down to 47%. Hence, node 3 represents a *dependability bottleneck* for topology 2. Guided by this result, we propose another slight modification to the topology, obtaining a new version (topology 3) shown in figure 9c, reducing both the number of child nodes of node 3 and the path length of nodes 8 and 11. In this case, looking at the *Topology 3* column in table 4, we can observe that the criticality on node 3 disappears, while reducing the overall number of critical events compared to topology 1. The benefits of this new topology are also visible in

Table 4. Classification of failure events for self-powered WSN (Topology 1,2 and 3)

	Failure Event			**Coverage**
	Topology 1	**Topology 2**	**Topology 3**	
Coverage $< 75\%$ (critical events)	*no event failure*	Stop(3) OR Disconnect(3,1)	*no event failure*	47%
	Stop(2) OR Stop(3) OR Disconnect(2,1) OR Disconnect(3,1)	*no event failure*	Stop(2) OR Stop(3) OR Disconnect(2,1) OR Disconnect(3,1)	54%
	no event failure	Stop(2) OR Disconnect(2,1)	*no event failure*	62%
	Stop(5) OR Stop(7) OR Disconnect(5,2) OR Disconnect(7,3)	*no event failure*	*no event failure*	70%
Coverage $\geq 75\%$	*no failure event*	Stop(6) OR Stop(7) OR Disconnect(6,3) OR Disconnect(7,3)	Stop(5) OR Stop(7) OR Disconnect(5,2) OR Disconnect(7,3)	77%
	no failure event	Stop(4) OR Stop(5) OR Disconnect(4,2) OR Disconnect(5,2)	*no failure event*	85%
	Stop(4) OR Stop(6) OR Stop(8) OR Stop(9) OR Stop(10) OR Stop(11) OR Stop(12) OR Stop(13) OR Disconnect(4,2) OR Disconnect(6,3) OR Disconnect(8,5) OR Disconnect(9,5) OR Disconnect(10,5) OR Disconnect(11,7) OR Disconnect(12,7) OR Disconnect(13,7)	Stop(8) OR Stop(9) OR Stop(10) OR Stop(11) OR Stop(12) OR Stop(13) OR Disconnect(8,5) OR Disconnect(9,5) OR Disconnect(10,5) OR Disconnect(11,7) OR Disconnect(12,7) OR Disconnect(13,7)	Stop(4) OR Stop(6) OR Stop(8) OR Stop(9) OR Stop(10) OR Stop(11) OR Stop(12) OR Stop(13) OR Disconnect(4,2) OR Disconnect(6,3) OR Disconnect(8,5) OR Disconnect(9,5) OR Disconnect(10,5) OR Disconnect(11,7) OR Disconnect(12,7) OR Disconnect(13,7)	93%

Table 5. Classification of failure events for MEDiSN (Topology 1, 2 and 3)

	Failure Event			**Coverage**
	Topology 1	**Topology 2**	**Topology 3**	
Coverage $< 75\%$ (critical events)	Stop(2) OR Disconnect(2,1)	*no event failure*	*no event failure*	10%
	Stop(3) OR Disconnect(3,2)	*no event failure*	*no event failure*	20%
	no event failure	*no event failure*	Stop(2) OR Disconnect(2,1)	40%
	no event failure	Stop(2) OR Disconnect(2,1)	*no event failure*	50%
	no event failure	Stop(3) OR Stop(4) OR Disconnect(3,1) OR Disconnect(4,2)	Stop(6) OR Disconnect(6,2)	60%
	Stop(5) OR Stop(6) OR Disconnect(5,3) OR Disconnect(6,3)	Stop(5) OR Stop(6) OR Disconnect(5,4) OR Disconnect(6,3)	Stop(3) OR Disconnect(3,1)	70%
Coverage $\geq 75\%$	Stop(4) OR Stop(7) OR Stop(8) OR Stop(9) OR Stop(10) OR Disconnect(4,3) OR Disconnect(7,5) OR Disconnect(8,5) OR Disconnect(9,6) OR Disconnect(10,6)	Stop(7) OR Stop(8) OR Stop(9) OR Stop(10) OR Disconnect(7,5) OR Disconnect(8,5) OR Disconnect(9,6) OR Disconnect(10,6)	Stop(4) OR Stop(5) OR Stop(7) OR Stop(8) OR Stop(9) OR Stop(10) OR Disconnect(4,2) OR Disconnect(5,3) OR Disconnect(7,3) OR Disconnect(8,6) OR Disconnect(9,6) OR Disconnect(10,6)	90%

table 3; specifically, topology 3 is able to improve the robustness in all cases with respect to topologies 1 and 2.

In a similar way, we exploit our approach to find better topologies for the MEDiSN case study. In this case we have observed that if we keep the same number of tree levels but reduce the number of leaf nodes (see the topology in Figure 10b), the robustness level decreases, as shown in the *Topology*

Table 6. Reasoning time of the ADVISES Tool for Self-powered WSN and MEDiSN considering a threshold value equal to 75%

Connection resiliency	Self-powered WSN		MEDiSN	
	Failure sequences	Elapsed Time (*s*)	Failure sequences	Elapsed Time (*s*)
1	24	1 500	18	600
2	552	9 720	306	2 160
3	12 144	32 580	4 896	2 820

2 column in table 3. Looking at table 5 (reporting the classification of critical events achieved for the MEDiSN WSN) we can note that, even if topology 2 is able to reduce the criticality of nodes 2 and 3, which are clearly two dependability bottlenecks for the original MEDiSN topology, it introduces new critical events (e.g. the failure of node 4). Moved by these results, we increase the number of leaf nodes and decrease the tree levels, obtaining a third topology (topology 3 in Figure 10c) able to reduce the number of critical events (see topology 3 column in table 5) and to improve the overall resiliency level in all cases (see topology 3 column in table 3).

In both the cases we have seen how the proposed approach is useful to precisely spot dependability bottlenecks and define more robust configurations. For both experiments, the ADVISES tool has performed a total of 161 460 reasonings on the specifications (114 480 for Self-powered WSN and 46 980 for MEDiSN). In particular, table 6 shows the number of the failure sequences and the related time spent by the ADVISES Tool to perform the reasoning in the self-powered WSN and in MEDiSN networks considering a coverage threshold value equal to 75%. In the worst case (12 144 different failure sequences tested for the Self-powered WSN) the tool takes about 9 hours to perform the evaluation on our commodity hardware. While this time could still be acceptable at design time (and it does not affect the conceptual validity of the approach), it clearly represents a practical limit, especially for large WSNs. This issue, and related solutions we are currently investigating, is further discussed in Section 8.

7.2.2. Runtime Verification

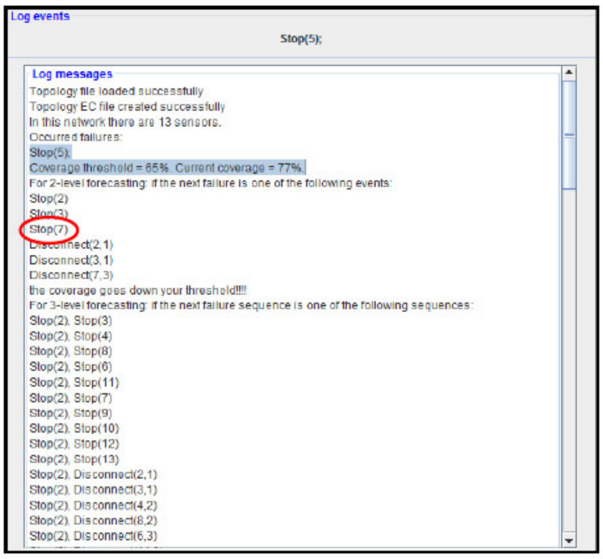
After the static analysis, we physically deploy the best topologies (topology 3 for both WSNs) at our labs to perform runtime verification.

For this purpose, we have designed and implemented a *system monitor* with the aim of detecting failure events from the real-world WSN. The monitor runs on a machine and listens for packets coming from all sensors trough the sink node of the WSN. The detection of events (such as the stop of a node) is performed assuming that each sensor sends packets periodically, with a known rate, which is common to several WSN applications. Hence, for every node, the monitor sets a timeout, which is reset each time the monitor receives a packet from the given node. If the timeout expires for a node X, the monitor sends a *Stop(X)* event to ADVISES (running in runtime mode). The use of time out may also detect temporary disconnections or delays. In this case, when packets from a node X are received again after a stop, the monitor sends a *Start(X)* event to ADVISES. Clearly, different failure detection approaches could be used as well, however this is not relevant for our experiment and out of the scope of the paper.

The monitor has been implemented as a java application running on a server (a Pentium 4 machine in our case) and connected via USB to a MIB520 Base station by Crossbow. As sensor nodes, we have adopted Iris Motes by Crossbow equipped with ZigBee RF Transceiver and TinyOS 2.0 operating system, running the *BlinkToRadio* application, just to perform a periodic sensing and sending of packets of all nodes to the sink.

Considering the deployment of third Self-powered WSN topology, we start the system monitor that is in communication with the ADVISES Tool.

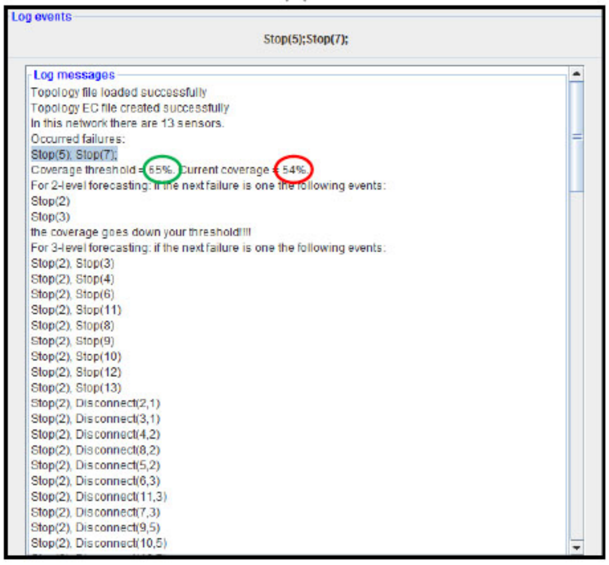
As a first test, we stop node 5. The failure event is detected by the system monitor at runtime (*elapsed time* $\approx$ *1 s*) and it is sent to ADVISES. Once received the failure event, ADVISES starts the



```
Log events
Stop(5);

Log messages
Topology file loaded successfully
Topology EC file created successfully
In this network there are 13 sensors.
Occurred failures:
Stop(5);
Coverage threshold = 65%. Current coverage = 77%.
For 2-level forecasting: if the next failure is one of the following events:
Stop(2)
Stop(3)
Stop(7)
Disconnect(2,1)
Disconnect(3,1)
Disconnect(7,3)
the coverage goes down your threshold!!!!
For 3-level forecasting: if the next failure sequence is one of the following sequences:
Stop(2), Stop(3)
Stop(2), Stop(4)
Stop(2), Stop(8)
Stop(2), Stop(6)
Stop(2), Stop(11)
Stop(2), Stop(7)
Stop(2), Stop(9)
Stop(2), Stop(10)
Stop(2), Stop(12)
Stop(2), Stop(13)
Stop(2), Disconnect(2,1)
Stop(2), Disconnect(3,1)
Stop(2), Disconnect(4,2)
Stop(2), Disconnect(8,2)
Stop(2), Disconnect(6,3)
```

(a) Node 5 failure



```
Log events
Stop(5);Stop(7);

Log messages
Topology file loaded successfully
Topology EC file created successfully
In this network there are 13 sensors.
Occurred failures:
Stop(5), Stop(7);
Coverage threshold = 65%. Current coverage = 54%.
For 2-level forecasting: if the next failure is one the following events:
Stop(2)
Stop(3)
the coverage goes down your threshold!!!!
For 3-level forecasting: if the next failure is one the following events:
Stop(2), Stop(3)
Stop(2), Stop(4)
Stop(2), Stop(6)
Stop(2), Stop(11)
Stop(2), Stop(8)
Stop(2), Stop(9)
Stop(2), Stop(10)
Stop(2), Stop(12)
Stop(2), Stop(13)
Stop(2), Disconnect(2,1)
Stop(2), Disconnect(3,1)
Stop(2), Disconnect(4,2)
Stop(2), Disconnect(8,2)
Stop(2), Disconnect(5,2)
Stop(2), Disconnect(6,3)
Stop(2), Disconnect(11,3)
Stop(2), Disconnect(7,3)
Stop(2), Disconnect(9,5)
Stop(2), Disconnect(10,5)
```

(b) Node 5 and 7 failures

Figure 11. ADVISES Tool in RV mode

reasoning and shows the result in the *Log message* panel as described in figure 11a (*metric computation time* $\approx$ *1 minute and 30 seconds*): coverage is equal to 77%. ADVISES also identifies potential critical nodes (*prediction* phase) that could compromise the robustness of the network; in fact, from figure 11a we can see that tool "advises" user (*prediction time* $\approx$ *4 minutes*) that if the next failure event that occurs is, for example, *Stop(7)*, then the network is not robust anymore (the coverage goes below the desired threshold, 65% in this test). To validate this suggestion, we stop also sensor node 7. From the log posted by the ADVISES tool (figure 11b), we observe that effectively the coverage decreases (54%) and goes under the desired threshold.

From the experiment, we can note that the adoption of the approach to monitor a real WSN at runtime is straightforward. Using the same formal specifications used at design time, the tool is able to tell at runtime what is the current dependability level of the WSN (e.g., coverage at 77%) and to pinpoint critical nodes (e.g., nodes 2, 3 and 7 in the example) that need to be maintained (e.g., by replacing batteries) or strengthened (e.g., by replicating them with other nodes) to avoid the whole mission of the WSN be compromised.

8. Conclusions and Future Work

This paper addressed the problem of the dependability assessment of WSNs with formal methods. Assessing the dependability of WSNs is a crucial task since this kind of networks are more and more used into critical application scenarios where the level of trust becomes an important factor; depending on application scenarios, different dependability requirements can be defined, such as, node lifetime, network resiliency, and coverage of the monitoring area. From a preliminary analysis it emerged the need of verifying a WSN at design time in order to increase the confidence about the robustness of designed solutions before putting it into operation and the need of monitoring a WSN during operation in order to avoid unexpected results or dangerous effects and thus to perform what in the literature is defined as *continuous monitoring*.

The research activity dealt with the definition of formal specifications used for the behavioral checking of WSN based systems at design and runtime phases; a set of correctness specifications applied to a generic WSN has been defined using event calculus as formal language since the behavior of a WSN can be characterized in terms of an event flow (e.g. a node turns on, a packet is sent, a node stops due to failure, etc.) and the event calculus formalism allowed to easily specify the system in terms of events.

This paper demonstrated that it is possible to assess the dependability of WSNs by means of the formal methods, in particular event calculus formalism, using the *narrative* it generates. By means of two case studies, we have shown how the adoption of a formal specification is helpful to deeply investigate the reasons of inefficiencies, in terms of the degree of dissatisfaction of given dependability requirements, and to suggest viable improvements to the design. The implementation of a tool, named ADVISES, has also shown how the approach can be easily adopted by technicians with no experience on formal methods, being the structural specification generated automatically and completely hidden to users. Finally, the paper described how the same specification can be adopted to perform continuous monitoring at runtime, once that the designed WSN gets implemented and deployed on the field.

However, the adoption of the approach also evidences a practical limit to be solved. We have considered topologies of WSNs adopted in typical critical scenarios, such as healthcare (with 7/15 nodes). During the experimental phase, we have also tested the correct functioning of the specification with topologies with more nodes (about 100) but the reasoning time becomes impractical on our commodity hardware (about 1 hour to test all the sequences with only 1 failure), due to the state space explosion problem. While this issue does not affect the conceptual validity of the approach and its use on typical critical WSNs, it undermines its practical adoption for large WSNs. To face this scalability problem, we are currently conceiving a method to divide a large topology in several sub-topologies (equivalent to WSN clusters), perform reasoning on each sub-topology in parallel and finally join the results taking in account the dependences between the sub-topologies.

We base our approach on sensors that are fixed (such as beacons) and with a established data routing reducing a topology like a spanning tree that is valid for a WSN. As future work we plan to extend the use of the specification also for mobile scenarios specifying further events that notify wireless sensor movements within clusters and from cluster to cluster.

Bibliography

1. Stankovic, J.A. Wireless sensor networks. *computer* **2008**, *41*, 92–95.
2. Avizienis, A.; Laprie, J.C.; Randell, B.; Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on* **2004**, *1*, 11–33.
3. Yick, J.; Mukherjee, B.; Ghosal, D. Wireless sensor network survey. Elsevier, 2008, Vol. 52, pp. 2292–2330.

4. Alemdar, H.; Ersoy, C. Wireless sensor networks for healthcare: A survey. *Computer Networks* **2010**, *54*, 2688–2710.
5. Coronato, A.; De Pietro, G. Formal Design of Ambient Intelligence Applications. *Computer* **2010**, *43*, 60 –68.
6. Augusto, J.C.; Zheng, H.; Mulvenna, M.D.; Wang, H.; Carswell, W.; Jeffers, W.P. Design and Modelling of the Nocturnal AAL Care System. ISAmI, 2011, pp. 109–116.
7. Aarts, E.; Wichert, R. Ambient intelligence. *Technology Guide* **2009**, pp. 244–249.
8. Coronato, A.; De Pietro, G. Formal Specification and Verification of Ubiquitous and Pervasive Systems. *ACM Trans. Auton. Adapt. Syst.* **2011**, *6*, 9:1–9:6.
9. Cinque, M.; Cotroneo, D.; Di Martinio, C.; Russo, S. Modeling and Assessing the Dependability of Wireless Sensor Networks. Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems; IEEE Computer Society: Washington, DC, USA, 2007; SRDS '07, pp. 33–44.
10. Woodcock, J.; Larsen, P.G.; Bicarregui, J.; Fitzgerald, J. Formal methods: Practice and experience. ACM, 2009, Vol. 41, p. 19.
11. Leucker, M.; Schallhart, C. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming* **2009**, *78*, 293 – 303.
12. Bakhouya, M.; Campbell, R.; Coronato, A.; Pietro, G.d.; Ranganathan, A. Introduction to special section on formal methods in pervasive computing. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* **2012**, *7*, 6.
13. Testa, A.; Coronato, A.; Cinque, M.; Augusto, J.C. Static Verification of Wireless Sensor Networks with Formal Methods. Signal Image Technology and Internet Based Systems (SITIS), 2012 Eighth International Conference on. IEEE, 2012, pp. 587–594.
14. Bondavalli, A.; Ceccarelli, A.; Falai, L.; Vadursi, M. A New Approach and a Related Tool for Dependability Measurements on Distributed Systems. *Instrumentation and Measurement, IEEE Transactions on* **2010**, *59*, 820–831.
15. Li, M.; Liu, Y. Underground coal mine monitoring with wireless sensor networks. *ACM Trans. Sen. Netw.* **2009**, *5*, 10:1–10:29.
16. Cinque, M.; Cotroneo, D.; Martino, C.D.; Russo, S.; Testa, A. AVR-INJECT: A tool for injecting faults in Wireless Sensor Nodes. IPDPS, 2009, pp. 1–8.
17. Cinque, M.; Cotroneo, D.; Di Martino, C.; Testa, A. An effective approach for injecting faults in wireless sensor network operating systems. Computers and Communications (ISCC), 2010 IEEE Symposium on. IEEE, 2010, pp. 567–569.
18. Shrestha, A.; Xing, L.; Liu, H. Infrastructure Communication Reliability of Wireless Sensor Networks. Dependable, Autonomic and Secure Computing, 2nd IEEE International Symposium on, 2006, pp. 250 –257.
19. Koushanfar, F.; Potkonjak, M.; Sangiovanni-Vincentelli, A. On-line fault detection of sensor measurements. Sensors, 2003. Proceedings of IEEE, 2003, Vol. 2, pp. 974 – 979 Vol.2.
20. Zhang, J.; Li, W.; Cui, D.; Zhao, X.; Yin, Z. The NS2-Based Simulation and Research on Wireless Sensor Network Route Protocol. Wireless Communications, Networking and Mobile Computing, 2009. WiCom '09. 5th International Conference on, 2009, pp. 1 –4.
21. Titzer, B.L.; Lee, D.K.; Palsberg, J. Avrora: scalable sensor network simulation with precise timing. Proceedings of the 4th international symposium on Information processing in sensor networks; IEEE Press: Piscataway, NJ, USA, 2005; IPSN '05.
22. Di Martino, C.; Cinque, M.; Cotroneo, D. Automated Generation of Performance and Dependability Models for the Assessment of Wireless Sensor Networks. *IEEE Trans. Comput.* **2012**, *61*, 870–884.
23. Heinzelman, W.; Chandrakasan, A.; Balakrishnan, H. Energy-efficient communication protocol for wireless microsensor networks. System Sciences, 2000. Proceedings of the 33rd Annual Hawaii International Conference on, 2000, p. 10 pp. vol.2.
24. Mini, A.F.; Nath, B.; Loureiro, A.A.F. A Probabilistic Approach to Predict the Energy Consumption in Wireless Sensor Networks. In IV Workshop de Comunicao sem Fio e Computao Mvel. So Paulo, 2002, pp. 23–25.
25. Kapitanova, K.; Son, S. MEDAL: A coMpact event description and analysis language for wireless sensor networks. Networked Sensing Systems (INSS), 2009 Sixth International Conference on, 2009, pp. 1–4.

26. Man, K.L.; Vallee, T.; Leung, H.; Mercaldi, M.; van der Wulp, J.; Donno, M.; Pastrnak, M. TEPAWSN - A tool environment for Wireless Sensor Networks. *Industrial Electronics and Applications, 2009. ICIEA 2009. 4th IEEE Conference on* **May**, pp. 730–733.
27. Boonma, P.; Suzuki, J. Moppet: A Model-Driven Performance Engineering Framework for Wireless Sensor Networks. *Comput. J.* **2010**, *53*, 1674–1690.
28. Shanahan, M. The Event Calculus Explained. *Lecture Notes in Computer Science* **1999**, *1600*, 409–430.
29. Ölveczky, P.; Meseguer, J. Specification and analysis of real-time systems using Real-Time Maude. *Fundamental Approaches to Software Engineering* **2004**, pp. 354–358.
30. Romadi, R.; Berbia, H. Wireless Sensor Network A specification method based on Reactive Decisional Agents. Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference on, 2008, pp. 1 –5.
31. Liang, Y.; Liu, R. Routing topology inference for wireless sensor networks. *SIGCOMM Comput. Commun. Rev.* **2013**, *43*, 21–28.
32. Zoumboulakis, M.; Roussos, G. Complex Event Detection in Extremely Resource-Constrained Wireless Sensor Networks; Kluwer Academic Publishers: Hingham, MA, USA, 2011; Vol. 16, pp. 194–213.
33. Bromuri, S.; Stathis, K. Distributed agent environments in the Ambient Event Calculus. Proceedings of the Third ACM International Conference on Distributed Event-Based Systems; ACM: New York, NY, USA, 2009; DEBS '09, pp. 12:1–12:12.
34. Blum, J.; Magill, E. Telecare service challenge: Conflict detection. Pervasive Computing Technologies for Healthcare (PervasiveHealth), 2011 5th International Conference on, 2011, pp. 502–507.
35. Kowalski, R.; Sergot, M. A logic-based calculus of events. *New Gen. Comput.* **1986**, *4*, 67–95.
36. Mueller, E.T. Automating commonsense reasoning using the event calculus. *Communications of the ACM* **2009**, *52*, 113–117.
37. Van Harmelen, F.; Lifschitz, V.; Porter, B. *Handbook Of Knowledge Representation*; Foundations of Artificial Intelligence, Elsevier, 2008.
38. Kim, T.W.; Lee, J.; Palla, R. Circumscriptive Event Calculus as Answer Set Programming. IJCAI, 2009, Vol. 9, pp. 823–829.
39. Hamadi, Y.; Jabbour, S.; Sais, L. ManySAT: a Parallel SAT Solver. *JSAT* **2009**, *6*, 245–262.
40. Mueller, E.T. DECReasoner. http://decreasoner.sourceforge.net **2005**.
41. Muller, E.T. Discrete Event Calculus Reasoner Documentation. http://decreasoner.sourceforge.net/csr/decreasoner.pdf **2008**.
42. Gungor, V.C.; Hancke, G.P. Industrial wireless sensor networks: Challenges, design principles, and technical approaches. *Industrial Electronics, IEEE Transactions on* **2009**, *56*, 4258–4265.
43. Testa, A. Dependability assessment of wireless sensor networks with formal methods. PhD thesis, PhD Thesis, Dipartimento di Informatica e Sistemistica, Universitá di Napoli Federico II, www. mobilab. unina. it/tesiDottorato. html, 2013.
44. Quwaider, M.; Biswas, S. DTN routing in body sensor networks with dynamic postural partitioning. *Ad Hoc Netw.* **2010**, *8*, 824–841.
45. Coronato, A.; De Pietro, G. Tools for the Rapid Prototyping of Provably Correct Ambient Intelligence Applications. *Software Engineering, IEEE Transactions on* **2012**, *38*, 975–991.
46. Cinque, M.; Coronato, A.; Testa, A. Dependable Services for Mobile Health Monitoring Systems. *IJACI* **2012**, *4*, 1–15.
47. Ying, H.; Schlo&#x0308;sser, M.; Schnitzer, A.; Scha&#x0308;fer, T.; Schla&#x0308;fke, M.E.; Leonhardt, S.; Schiek, M. Distributed Intelligent Sensor Network for the Rehabilitation of Parkinson&#x0027;s Patients. *Trans. Info. Tech. Biomed.* **2011**, *15*, 268–276.
48. Wu, C.H.; Tseng, Y.C. Data Compression by Temporal and Spatial Correlations in a Body-Area Sensor Network: A Case Study in Pilates Motion Recognition. *IEEE Transactions on Mobile Computing* **2011**, *10*, 1459–1472.
49. Ko, J.; Lim, J.H.; Chen, Y.; Musvaloiu-E, R.; Terzis, A.; Masson, G.M.; Gao, T.; Destler, W.; Selavo, L.; Dutton, R.P. MEDiSN: Medical emergency detection in sensor networks. *ACM Trans. Embed. Comput. Syst.* **2010**, *10*, 11:1–11:29.

50. Fernández-López, H.; Afonso, J.A.; Correia, J.; Simões, R. HM4All: a vital signs monitoring system based in spatially distributed zigBee networks. Pervasive Computing Technologies for Healthcare (PervasiveHealth), 2010 4th International Conference on-NO PERMISSIONS. IEEE, 2010, pp. 1–4.
51. Hande, A.; Polk, T.; Walker, W.; Bhatia, D. Self-powered wireless sensor networks for remote patient monitoring in hospitals. *Sensors* **2006**, *6*, 1102–1117.
52. Fariborzi, H.; Moghavvemi, M. Architecture of a Wireless Sensor Network for Vital Signs Transmission in Hospital Setting. Proceedings of the 2007 International Conference on Convergence Information Technology; IEEE Computer Society: Washington, DC, USA, 2007; ICCIT '07, pp. 745–749.
53. Qiu, Y.; Zhou, J.; Baek, J.; Lopez, J. Authentication and Key Establishment in Dynamic Wireless Sensor Networks. *Sensors* **2010**, *10*, 3718–3731.
54. Chipara, O.; Lu, C.; Bailey, T.C.; Roman, G.C. Reliable clinical monitoring using wireless sensor networks: experiences in a step-down hospital unit. Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems; ACM: New York, NY, USA, 2010; SenSys '10, pp. 155–168.